Technical Refactoring Blueprint: Decomposition of the 'Player.ts' Monolith in *Binky Medieval Showdown*

1. Executive Summary and Architectural Audit

The ongoing development of *Binky Medieval Showdown* has reached a critical inflection point where architectural scalability is no longer a luxury but a necessity. The current implementation of the primary player entity, encapsulated within Player.ts, has exhibited the classic symptoms of a "God Object"—an anti-pattern where a single class assumes excessive responsibilities, ranging from input polling and physics management to animation state control and UI synchronization.[1] This monolithic design creates significant friction for implementing advanced combat mechanics, specifically the differentiation between the Knight (Tank) and Ninja (Evasive) archetypes.[1] To facilitate the desired depth in combat mechanics and ensure the codebase remains maintainable, a rigorous refactoring plan is required.

This report provides a comprehensive technical roadmap to dismantle the Player.ts monolith. The proposed architecture transitions the game from a manager-heavy, imperative model to a decoupled, component-driven system. This transformation relies on three specific architectural pillars: a robust, logic-driving Finite State Machine (FSM) [2]; a Command Pattern-based Input Manager with buffering capabilities [4]; and the standardization of the "Skill Object" pattern, derived from the existing Ogre enemy implementation, to handle combat actions polymorphically.[1]

1.1 The Pathology of the Player.ts Monolith

A deep static analysis of the Player.ts file reveals a high degree of coupling and cohesion issues that typify the God Object anti-pattern. While the class nominally employs a StateMachine property, its usage is superficial, serving more as a label for the current animation rather than a driver of game logic.[1] The Player class retains control over critical state variables, input interpretation, and timer management, effectively bypassing the benefits of the FSM it instantiates.

Critical Architectural Flaws:

- **Logic Leakage and Switch Statements:** The method performSecondarySkill is a primary offender, containing conditional logic (switch statements or if-else blocks) to determine behavior based on the archetype (e.g., executing Shield Bash for Tanks vs. Shadow Dash for Evasive types).[1] This violates the Open/Closed Principle; adding a new class requires modifying the core Player file, increasing the risk of regression bugs.
- **Temporal Coupling:** The Player class directly manages timers for state transitions

using scene.time.delayedCall. For instance, the transition back to IDLE after a skill execution is hard-coded into the player's update logic, tightly binding the visual representation (animation duration) to the logical state (control lock).[1] This makes tuning gameplay feel—such as "canceling" animations—extremely difficult.

- **Input Polling Dependency:** The update loop performs direct polling of the inputManager (e.g., this.inputManager.isActionJustPressed). This couples the frame rate to input responsiveness and makes implementing features like "input buffering" (registering a button press slightly before a state allows it) effectively impossible without cluttering the update loop with complex boolean flags.[1]
- **State Variable Centralization:** Critical flags such as isAttacking, isInvulnerable, and lastAttackTime reside on the Player instance rather than within specific states or components. This forces the Player class to act as the gatekeeper for all interactions, preventing states from being self-contained units of logic.[1]

The proposed solution involves stripping Player.ts of these responsibilities, reducing it to a lightweight Context object that holds data (stats, sprite) and delegates behavior to specialized systems.

---

2. Theoretical Framework: The Finite State Machine (FSM)

The first pillar of this refactoring effort is the elevation of the Finite State Machine from a passive observer to an active controller. In the current architecture, the FSM is "hybrid," meaning the Player class handles the bulk of the logic and simply tells the FSM to update its current state label.[1] The target architecture requires a "Pure FSM," where the State classes themselves contain the logic for input handling, physics application, and transitions.

2.1 Moving from Labels to Logic

In a component-driven game architecture, a State should be more than an enum wrapper. It must be a distinct class that encapsulates the behavior of the entity during that specific mode of operation.[2]

The Passive FSM (Current):

- **Input:** Player.update() reads keys.
- **Logic:** Player.update() checks if (key.isDown && state === IDLE).
- **Action:** Player calls player.setVelocity().
- **FSM:** Player calls stateMachine.transition(RUN).

The Active FSM (Target):

- **Input:** InputManager pushes commands to InputBuffer.
- **Logic:** Player.update() calls stateMachine.update().
- **Action:** IdleState.update() peeks at buffer, sees MoveCommand, transitions to MoveState.

- **Execution:** MoveState.enter() applies velocity and plays animation.

This inversion of control is critical. It removes the "cyclomatic complexity" from the Player class, distributing it across small, testable State classes.[3]

2.2 Hierarchical State Machine (HFSM) Design

Given the complexity of actions described in the Player.ts analysis—CASTING_SECONDARY, CHANNELING_SPECIAL, ATTACK_PRIMARY—a flat FSM can lead to code duplication. For example, the logic for "taking damage" or "falling off a cliff" applies to almost every state. To avoid implementing handleDamage() in every single class, we will utilize a Hierarchical State Machine approach.[9]

We define a root GroundedState that handles common physics interactions (friction, gravity) and interruptions (damage). Sub-states like Idle, Run, and Attack inherit from this.

Proposed Class Hierarchy:

- **AbstractState**: Defines the contract (enter, update, exit, handleInput).
    - **GroundedState**: Handles gravity checks, friction, and damage interrupts.
        - **IdleState**: Zero velocity, waits for input.
        - **MoveState**: Applies movement velocity based on input vector.
        - **AttackState**: Locks movement, spawns Skill Objects.
    - **AirborneState**: Handles air control (if applicable), landing logic.
    - **DashState** (Ninja): Overrides physics processing to ignore friction/gravity for fixed duration.

This structure adheres to the "Open/Closed Principle".[2] If a new mechanic like "Swimming" is introduced, a new SwimmingState branch is added without touching the existing movement logic.

2.3 State-Encapsulated Logic and Cleanup

A common source of bugs in the current monolith is "variable leakage," where a flag like isInvulnerable is set to true during a dash but fails to reset if the dash is interrupted. In the new FSM, the DashState is exclusively responsible for this flag.[10]

- **DashState.enter()**: Sets player.stats.isInvulnerable = true.
- **DashState.exit()**: Sets player.stats.isInvulnerable = false.

Because the FSM guarantees that exit() is called on state transition (even if interrupted by a stun or death event), the system becomes robust against invalid states. This creates a "safety harness" around complex mechanics like the Ninja's evasion abilities.[1]

---

3. The Command Pattern and Input Buffering

The second pillar of the refactoring is decoupling the Player's intent from the hardware inputs via the **Command Pattern**. Currently, the Player class is tightly coupled to the InputManager, polling specific keys every frame.[1] This "direct polling" is brittle; if the game drops a frame or the player presses a button 10ms before an animation ends, the input is lost, making the game

feel unresponsive.
To achieve the "Showdown" feel—responsive, snappy combat—we must implement an Input Buffer powered by Commands.[4]

3.1 Reification of Input

The Command Pattern involves "reifying" (turning into an object) method calls. Instead of if (key.isDown) jump(), the Input Manager generates a JumpCommand object.
Advantages of Reification:

1. **Buffering:** Objects can be stored in a queue (Array) and processed later.
2. **Remapping:** The Command object represents the *intent* (ATTACK), not the hardware (KEY_SPACE). The Input Manager handles the mapping.
3. **AI Control:** An AI agent can push AttackCommand objects into the queue just like a keyboard listener, unifying the logic for Players and Enemies.[5]

3.2 The Input Buffer Implementation

The Input Buffer is a short-lived queue (FIFO) that holds commands for a specific time window (e.g., 150ms). This technique allows for "Coyote Time" (jumping just after leaving a platform) and pre-buffering attacks during the recovery frames of a previous move.[7]
Logic Flow:

1. **Event:** User presses 'Space'.
2. **InputManager:** Maps 'Space' to PrimarySkillCommand.
3. **Buffer:** Adds PrimarySkillCommand to player.inputBuffer with a timestamp.
4. **Update Loop:** Player.update runs stateMachine.update().
5. **State Logic:** The active state (e.g., AttackState) is finishing. In its update method, it peeks at the buffer. It sees PrimarySkillCommand.
6. **Transition:** Because the animation is >90% complete, the state consumes the command and transitions to AttackState (combo chain).

This effectively decouples the *time* the input occurred from the *time* it is executed, smoothing over the rigid frame boundaries of the game loop.[11]

3.3 TypeScript Implementation Strategy

The implementation will utilize TypeScript interfaces to enforce strict typing on commands.

TypeScript

```
export enum ActionType {
  MOVE,
  ATTACK_PRIMARY,
  ATTACK_SECONDARY,
  INTERACT
}
```

```
export interface ICommand {
   type: ActionType;
   timestamp: number;
   payload?: any; // e.g., Vector2 for movement
}
```

The InputManager in Player.ts will be replaced by a CommandReceiver. The actual InputManager will act as a Scene-level system (or global plugin) that emits events or pushes commands to the relevant actors.[5] This aligns with best practices for Phaser 3, treating input as scene state rather than entity state.[12]

---

4. The Skill Object Pattern: Learning from the Ogre

The third and most transformative pillar is the standardizing of combat logic using the **Skill Object Pattern**. Analysis of Enemy.ts reveals that enemies, particularly the Ogre, utilize specialized classes like MeleeAttack to handle their offensive actions.[1] This stands in stark contrast to the Player.ts class, which likely hard-codes hitbox creation logic inside performAttack.

4.1 Deconstructing the Ogre's Mechanics

The "Ogre" is a critical case study because it represents a complex, multi-stage attack that is decoupled from the main actor sprite.
Key Mechanics of the Ogre [1]:

- **Independent Hitbox Calculation:** The Ogre's attack range is dynamic: spriteRadius + 100 + 20. This implies the existence of a hitbox that is distinct from the Ogre's physical collision body.
- **Visual/Logic Separation:** The Ogre has an "Attack Duration" of 1.1 seconds, composed of a 0.8s animation and a 0.3s "bone slam" effect. This separation suggests the attack has a lifecycle that extends beyond the mere playback of a sprite animation.
- **Event-Driven Cleanup:** The Ogre uses animationcomplete listeners to revert to a walking state, ensuring the logic stays synchronized with the visuals.

4.2 The SkillObject Architecture

We will formalize this pattern into a generic SkillObject system for the player. A SkillObject is a Phaser.GameObjects.GameObject (often a Container or Zone) that is spawned by a Skill and exists in the world to perform a specific combat function.[6]
Anatomy of a Skill Object:

1. **The Data (Skill):** A lightweight class (ScriptableObject-style) containing metadata: Cooldown, Damage, Resource Cost, Icon.
2. **The Effector (SkillObject):** The physical entity in the scene.

    - **Telegraph:** Visual indicators (e.g., the Archer's ArrowIndicator [1]).
    - **Hitbox:** A Phaser Zone or Arcade.Body used for overlap checks.
    - **Visuals:** Particles, sprites, or shaders (e.g., the "Bone Slam").

Polymorphic Execution:
Instead of Player.ts having a switch (archetype), the Player will hold a Loadout:

- primarySkill: Skill
- secondarySkill: Skill

When the input command comes in, the Player simply calls this.loadout.primary.activate(this).

- If the player is a **Knight**, this instantiates a SwordSlashObject (creates hitbox in front, plays sound).
- If the player is a **Ninja**, this instantiates a KunaiProjectile (spawns physics sprite, sets velocity).

The Player class is now completely agnostic to the archetype mechanics. It doesn't know if it's swinging a sword or throwing a dagger; it only knows it executed a skill.[13]

4.3 Implementing Ogre-Style Features for Players

To bring the Player up to parity with the Ogre, we must adopt the **Delayed Hitbox** and **Animation Locking** patterns.
Delayed Hitbox (The "Bone Slam" Effect):
The Knight's "Shield Bash" should not apply damage the instant the button is pressed. It should match the animation's impact frame.

- **Implementation:** The ShieldBashSkill class will listen for the ANIMATION_UPDATE event or use a delayedCall. On the specific frame (e.g., frame 4), it *spawns* the ShieldBashObject (the hitbox).
- **Benefit:** This creates "weight" in combat. It also allows the enemy to interrupt the player during the wind-up, adding strategic depth.[1]

Animation Locking:
Just as the Ogre is locked for 1.1s, the Player's AttackState will enforce a lock. The Skill definition will return a duration value. The AttackState will remain active for that duration, ignoring MoveCommands (unless a specific "Cancel" mechanic is implemented, like the Ninja's Dash).1

---

5. Integration: Data Flow and System Interaction

The refactoring is not just about creating new classes; it's about defining how data flows between them. The proposed architecture creates a unidirectional flow of control.

5.1 The Control Loop

1. **Input Phase:**
    - Browser Events (Keydown) $\rightarrow$ **Scene Input Plugin** $\rightarrow$ **InputManager**.

    - **InputManager** converts Raw Input $\rightarrow$ **Command** (e.g., AttackCommand).
    - Command is pushed to **InputBuffer**.
2. **Update Phase (Player):**
    - Player.update() executes.
    - **CooldownManager** ticks down.
    - StateMachine.update() executes.
3. **Decision Phase (State):**
    - Current State (IdleState) checks InputBuffer.peek().
    - Finds AttackCommand.
    - Checks logic: CanAttack? (e.g., not stunned, cooldown ready).
    - If valid, StateMachine transitions to AttackState.
4. **Execution Phase (State & Skill):**
    - AttackState.enter() calls player.loadout.primary.activate().
    - **Skill** spawns **SkillObject** (e.g., MeleeHitbox) into the Scene.
    - **SkillObject** registers itself with the Physics System (e.g., EnemySystem collision groups).
5. **Resolution Phase (Physics):**
    - Phaser Arcade Physics detects overlap between **SkillObject** and **Enemy**.
    - SkillObject.onHit(enemy) is triggered.
    - Damage is applied to Enemy stats.
    - **Visuals** (Particles) are spawned at impact point.

5.2 Decoupling UI and Systems

Currently, Player.ts manages its own Health Bar and Skill Tree toggling.[1] In the refactored architecture, the Player should simply emit events.

- player.events.emit('healthChanged', newValue)
- player.events.emit('openSkillTree')

The GameScene or a dedicated UIManager listens for these events and handles the UI updates. This separates the logic (Player) from the view (UI), preventing the Player class from needing references to Text objects and Graphics containers.[1][2]

---

6. Implementation Roadmap

This transformation cannot happen in a single commit. A phased approach is necessary to maintain a working build throughout the refactoring process.

Phase 1: Input System Extraction

- **Goal:** Replace polling in Player.update with Command processing.
- **Steps:**
    1. Create src/input/Command.ts and src/input/InputBuffer.ts.
    2. Refactor InputManager to emit Commands instead of setting boolean flags.

3. Modify Player.ts to accept an InputBuffer in its constructor.
4. Update Player.update to log commands instead of moving. (Verification).
5. Restore movement logic using if (buffer.contains(MoveCommand)).

Phase 2: State Machine Solidification

- **Goal:** Move movement and idle logic out of Player.ts.
- **Steps:**
  1. Create src/states/AbstractState.ts with the new, active interface.
  2. Implement IdleState and MoveState, moving the setVelocity logic from Player.ts into these classes.[3]
  3. Refactor Player.ts to delegate the update loop to stateMachine.update().
  4. Verify that basic locomotion (Run/Idle) works identically to before.

Phase 3: Skill Object Foundation

- **Goal:** Standardize the concept of an attack.
- **Steps:**
  1. Define abstract Skill and SkillObject classes.
  2. Refactor the *existing* Ogre logic in Enemy.ts to use this new system (Refactoring by "paralysis"). This proves the system works for the complex Ogre case first.[1]
  3. Ensure MeleeAttack.ts extends SkillObject.

Phase 4: Player Combat Refactor

- **Goal:** Remove archetype switches from Player.ts.
- **Steps:**
  1. Create ShieldBashSkill and ShadowDashSkill extending the base Skill class.
  2. Create a SkillLoadout class on the Player.
  3. Initialize the Loadout based on the Archetype config.
  4. Replace performSecondarySkill in Player.ts with loadout.secondary.activate().
  5. Move the specific hitboxes and physics logic into the new Skill classes.

Phase 5: Cleanup and Optimization

- **Goal:** Remove dead code.
- **Steps:**
  1. Delete isAttacking, lastAttackTime, and other flags from Player.ts (now handled in Skill or AttackState).
  2. Verify EnemySystem collision logic works with the new Player Skill Objects (Player attacks hitting Enemies).
  3. Optimize object pooling for high-frequency skills (e.g., Ninja projectiles).

---

7. Comparative Analysis: Current vs. Proposed

Feature	Current Implementation	Proposed Architecture	Impact
Logic Control	Player.update (Monolith)	State.update (Distributed)	Reduces class size, improves readability.
Input Handling	Direct Polling (isDown)	Buffered Commands	Enables "Coyote Time" and precise combos.
Archetypes	Hard-coded switch statements	Polymorphic SkillLoadout	Infinite extensibility for new classes.
Hitboxes	Coupled to Sprite	Decoupled SkillObjects	Accurate hit detection, complex attack shapes.
State Flags	Public properties on Player	Encapsulated in State	Eliminates invalid states (e.g., moving while attacking).
Enemy Parity	Different logic for Player/Enemy	Unified SkillObject system	Easier to implement PvP or AI-vs-AI modes.

8. Detailed Code Specifications

8.1 The State Machine Infrastructure

The following TypeScript definitions outline the robust FSM structure.

TypeScript

```
// src/game/fsm/StateMachine.ts
export class StateMachine {
    private currentState: State;
    private states: Map<string, State> = new Map();

    public addState(key: string, state: State): void {
        this.states.set(key, state);
    }

    public transition(key: string, data?: any): void {
        if (this.currentState) {
            this.currentState.exit();
        }
        this.currentState = this.states.get(key)!;
        this.currentState.enter(data);
    }
```

```
    public update(time: number, delta: number): void {
        if (this.currentState) {
            this.currentState.update(time, delta);
        }
    }

    public handleInput(command: Command): void {
        if (this.currentState) {
            this.currentState.handleInput(command);
        }
    }
}
```

8.2 The Command-Based Input Buffer

TypeScript

```
// src/game/input/InputBuffer.ts
export class InputBuffer {
    private commands: Command =;
    private readonly bufferWindow: number = 200; // ms

    public add(cmd: Command): void {
        this.commands.push(cmd);
        // Prune old commands
        const now = Date.now();
        this.commands = this.commands.filter(c => now - c.timestamp < this.bufferWindow);
    }

    public consume<T extends Command>(type: ActionType): T | null {
        const index = this.commands.findIndex(c => c.type === type);
        if (index!== -1) {
            return this.commands.splice(index, 1) as T;
        }
        return null;
    }
}
```

8.3 The Ogre-Style Skill Object

This implementation mimics the "bone slam" delay and independent hitbox logic.

TypeScript

```
// src/game/skills/types/ShieldBash.ts
export class ShieldBash extends Skill {
   activate(player: Player): void {
      // 1. Lock Player State
      player.stateMachine.transition(PlayerState.ATTACK_LOCK, { duration: 500 });

      // 2. Play Animation
      player.playAnimation('SHIELD_BASH');

      // 3. Delayed Hitbox Spawn (The Ogre Pattern)
      player.scene.time.delayedCall(200, () => {
         const hitbox = new SkillObject(player.scene, player.x, player.y);
         hitbox.setSize(60, 40); // Rectangular Bash
         // Offset hitbox in front of player based on facing
         const offset = player.flipX? -40 : 40;
         hitbox.setPosition(player.x + offset, player.y);

         // Add to physics system for 100ms active frame
         player.scene.physics.add.existing(hitbox);
         player.scene.time.delayedCall(100, () => hitbox.destroy());
      });
   }
}
```

9. Conclusion

The dismantling of the Player.ts monolith is a foundational requirement for the future of *Binky Medieval Showdown*. By aligning the Player's architecture with the existing, robust patterns found in the Ogre enemy logic, the game gains consistency and scalability. The shift to a Command Pattern ensures the gameplay meets the responsiveness expectations of the genre, while the Active FSM ensures that the code remains clean and organized as new mechanics are introduced. This refactoring plan transforms the Player from a collection of hard-coded behaviors into a flexible, data-driven entity capable of supporting the "Advanced Combat Mechanics" envisioned for the project.

10. Tables and Data Structures

Table 1: State Responsibilities Matrix

State	Input Focus	Physics Behavior	Interrupt Conditions
IdleState	Listens for MOVE, ATTACK	Velocity = 0, High Friction	Damage, Knockback
MoveState	Listens for ATTACK, STOP	Velocity = Stats.Speed, Facing updates	Damage, Knockback
AttackState	Inputs Ignored (Locked)	Velocity = 0 (or lunge impulse)	Death only (Uninterruptible)
DashState	Inputs Ignored	Velocity = High (Fixed Vector)	None (Invulnerable)

Table 2: Input Command Mapping

Hardware Input (Default)	Action Enum	Command Object	Context
W, A, S, D / Stick	MOVE	MoveCommand(Vector2)	Continuous
Space / A Button	JUMP/DASH	DashCommand	Instant
Left Click / X Button	ATTACK_PRIMARY	SkillCommand(Slot.PRIMARY)	Buffered
Right Click / B Button	ATTACK_SECONDARY	SkillCommand(Slot.SECONDARY)	Buffered

Table 3: Refactoring Metrics & Targets

Metric	Current Value (Est.)	Target Value	Reason
Player.ts LOC	~800+ lines	< 200 lines	Delegation to sub-systems.
switch statements	High complexity	0 (Polymorphic)	Replaced by Skill Objects.
Input Latency	Frame-dependent	Buffered (150ms)	Command Pattern implementation.
New Class Cost	High (Edit Player.ts)	Low (New Config)	Data-driven loadouts.

Works cited

1. Player.ts
2. A Guide to the State Design Pattern in TypeScript and Node.js with Practical Examples | by Robin Viktorsson | Medium, accessed December 4, 2025, https://medium.com/@robinviktorsson/a-guide-to-the-state-design-pattern-in-typescript-and-node-js-with-practical-examples-20e92ff472df
3. State Pattern for Character Movement in Phaser 3 - Ourcade Blog, accessed December 4, 2025, https://blog.ourcade.co/posts/2020/state-pattern-character-movement-phaser-3/

4. Command · Design Patterns Revisited - Game Programming Patterns, accessed December 4, 2025, https://gameprogrammingpatterns.com/command.html
5. GaryStanton/phaser3-merged-input: A Phaser 3 plugin to map input from keyboard & gamepad to player actions - GitHub, accessed December 4, 2025, https://github.com/GaryStanton/phaser3-merged-input
6. [Phaser 3 – WIP] Fantasy Sphere RPG - Showcase, accessed December 4, 2025, https://phaser.discourse.group/t/phaser-3-wip-fantasy-sphere-rpg/967
7. Handling Inputs in Phaser 3: Part 1: Humble Beginnings | Coding into the Void, accessed December 4, 2025, https://blog.khutchins.com/posts/phaser-3-inputs-1/
8. State · Design Patterns Revisited - Game Programming Patterns, accessed December 4, 2025, https://gameprogrammingpatterns.com/state.html
9. Using finite state machines to build scalable player controllers - Michael Bitzos, accessed December 4, 2025, https://michaelbitzos.com/devblog/fsm-player-controllers
10. State Pattern for Changing AI and Player Control in Phaser 3 - Ourcade Blog, accessed December 4, 2025, https://blog.ourcade.co/posts/2020/state-pattern-ai-player-control-phaser-3/
11. Simulate a keyboard input delay with Phaser (why on earth? You'll see) | Emanuele Feronato, accessed December 4, 2025, https://emanueleferonato.com/2025/02/14/simulate-a-keyboard-input-delay-with-phaser-why-on-earth-youll-see/
12. What are Phaser 3 bad/best practices?, accessed December 4, 2025, https://phaser.discourse.group/t/what-are-phaser-3-bad-best-practices/5088
13. Decoupling the entity from whatever controls it - Game Development Stack Exchange, accessed December 4, 2025, https://gamedev.stackexchange.com/questions/80334/decoupling-the-entity-from-whatever-controls-it
14. Composing game objects (kinda like ECS) - Phaser 3 - Discourse, accessed December 4, 2025, https://phaser.discourse.group/t/composing-game-objects-kinda-like-ecs/1676
15. Events - What is Phaser?, accessed December 4, 2025, https://docs.phaser.io/phaser/concepts/events