Pyramid Poker Online - Data Models and Standards

This document defines the canonical data formats (schemas) used throughout the Pyramid Poker Online application. All modules should conform to these standard models to ensure consistency and interoperability.

Core Principle

Single Source of Truth: Each data type (Card, Hand, Arrangement) has ONE canonical format that all modules must use. This eliminates format conversion complexity and ensures consistency across the application.

Data Models

Card Model (Standard Schema)

The universal card format used by all modules:

```javascript
{
    id: string,          // Unique identifier (e.g., "A♠_1", "6♥_2")
    rank: string,        // Card rank: "2"-"10", "J", "Q", "K", "A"
    suit: string,        // Card suit: "♠", "♥", "♦", "♣" (or "" for wild)
    value: number,       // Numeric value for comparison (2-14, A=14)
    isWild: boolean,     // True if this is a wild card
    wasWild?: boolean    // Optional: True if originally wild but substituted
}
```

Required Properties: `id`, `rank`, `suit`, `value`, `isWild`
Optional Properties: `wasWild`

Hand Model (Standard Schema)

The universal hand format used by all modules:

```javascript
{
    cards: Array<Card>,              // Array of Card objects in this hand
    handType: string,                // "Pair", "Straight", "Four of a Kind", etc.
    cardCount: number,               // Number of cards in hand (3, 5, 6, 7, 8)
    handStrength: Object,            // Full evaluation result from card-evaluation.js
    hand_rank: Array<number>,        // Proper ranking tuple for comparison
    strength: number,                // Numeric strength for sorting
    validPositions: Array<string>,   // ["front", "middle", "back"] - where hand can be placed
    isIncomplete: boolean,           // True if hand needs kickers to be legal
    kickersNeeded?: Object,          // Kickers needed for each position (if incomplete)
    positionScores?: Object          // Points if win for each valid position
}
```

Arrangement Model (Standard Schema)

The universal arrangement format used by all modules:

```javascript
{
    back: Hand,          // Back hand (strongest, 5-8 cards)
    middle: Hand,        // Middle hand (medium strength, 5-7 cards)
    front: Hand,         // Front hand (weakest, 3 or 5 cards)
    score: number,       // Total expected score for this arrangement
    isValid: boolean,    // True if arrangement follows game rules
    statistics?: Object  // Optional: performance metrics from generation
}
```

```

## Implementation Strategy

### Phase 1: Incremental Adoption (Current)
- **Document standards** (this document)
- **Create converters** when format mismatches occur
- **Fix immediate compatibility issues** without breaking working code

### Phase 2: Module Migration
- **Gradually update modules** to natively use standard schemas
- **Add validation functions** to ensure format compliance
- **Update documentation** for each converted module

### Phase 3: Cleanup
- **Remove converters** when no longer needed
- **Enforce standards** in new code
- **Complete migration** to universal formats

## Converter Functions

When format conversion is necessary during migration:

```javascript
// Convert various formats to Card Model
function convertToCardModel(cardObjects) { ... }

// Convert various formats to Hand Model
function convertToHandModel(handObjects) { ... }

// Convert various formats to Arrangement Model
function convertToArrangementModel(arrangementObjects) { ... }
```

Module Compliance Status

✅ Compliant Modules
- `hand-detector.js` - Uses Card Model, produces Hand Model
- `best-arrangement-generator.js` - Uses Hand Model, produces Arrangement Model
- `card-evaluation.js` - Uses Card Model

🔄 Partially Compliant Modules
- `auto-arrange.js` - Uses custom card format, needs Card Model conversion
- `one-wild-brute-force-from-cards.js` - Receives custom format, needs conversion

❌ Non-Compliant Modules
- _(To be identified during migration)_

Validation Functions

```javascript
// Validate Card Model compliance
function validateCardModel(card) {
    const required = ['id', 'rank', 'suit', 'value', 'isWild'];
    return required.every(prop => card.hasOwnProperty(prop));
}

// Validate Hand Model compliance
function validateHandModel(hand) {
    const required = ['cards', 'handType', 'cardCount', 'handStrength', 'validPositions'];
    return required.every(prop => hand.hasOwnProperty(prop));
}

// Validate Arrangement Model compliance
function validateArrangementModel(arrangement) {
```
```

```
    const required = ['back', 'middle', 'front', 'score', 'isValid'];
    return required.every(prop => arrangement.hasOwnProperty(prop));
}
```

Benefits

1. **Consistency** - All modules use the same data formats
2. **Maintainability** - Changes to data structure happen in one place
3. **Debugging** - Easier to trace data flow when format is consistent
4. **Testing** - Standardized test data across all modules
5. **Documentation** - Clear expectations for each data type
6. **Performance** - No unnecessary format conversions

Future Enhancements

- **TypeScript definitions** for compile-time type checking
- **JSON Schema validation** for runtime format validation
- **Automated testing** to ensure all modules comply with standards
- **Code generation** tools to create standard objects

Last Updated: July 6, 2025
Status: Phase 1 - Incremental Adoption