JUCE DAW コーディングルール

対象読者: 開発者
バージョン: 1.0.0
最終更新日: 2025年6月19日

目次

1. はじめに
2. 命名規則
3. フォーマットとスタイル
4. コメントとドキュメンテーション
5. エラーハンドリング
6. パフォーマンスと最適化
7. メモリ管理
8. スレッドセーフティ
9. JUCE特有の規約
10. Tracktion Engine特有の規約
11. AI統合コードの規約
12. テストコードの規約
13. コードレビュー
14. ツールと自動化

---

1. はじめに

このドキュメントは、AI統合DAWプロジェクトにおけるC++コードの品質、一貫性、保守性を確保するためのコーディングルールを定義します。すべての開発者は、このルールに従ってコードを記述する必要があります。

1.1 目的

- **コードの可読性向上**: 誰でも理解しやすいコードを目指す
- **保守性の確保**: 将来の変更や拡張を容易にする
- **バグの低減**: 一貫したルールによる潜在的なバグの防止
- **チーム開発の効率化**: 共通の規約によるコミュニケーションコスト削減

1.2 適用範囲

このルールは、プロジェクト内のすべてのC++コード（アプリケーションロジック、UI、オーディオ処理、AI統合、テストコードなど）に適用されます。

1.3 基本原則

- **明確性**: コードは意図が明確であること
- **簡潔性**: 不必要な複雑さを避けること
- **一貫性**: プロジェクト全体でスタイルを統一すること
- **安全性**: メモリリークや競合状態などの問題を避けること
- **効率性**: パフォーマンスを意識したコードを記述すること

---

2. 命名規則

2.1 一般的な規則

- **言語**: 識別子は英語を使用する
- **明確性**: 名前は役割や機能を明確に表すこと
- **略語**: 一般的に認知されている略語（例: UI, API, URL）以外は避ける
- **予約語**: C++の予約語やJUCEの主要クラス名との衝突を避ける

2.2 ケーススタイル

- **クラス名、構造体名、列挙型名**: `PascalCase` (例: `AudioEngine`, `MidiNoteData`)
- **関数名、メソッド名**: `camelCase` (例: `processAudioBlock`, `initializeGui`)
- **変数名（ローカル、メンバ）**: `camelCase` (例: `audioBuffer`, `m_sampleRate`)
- **定数名、列挙子名**: `UPPER_SNAKE_CASE` (例: `MAX_BUFFER_SIZE`, `PlaybackState::Playing`)
- **マクロ名**: `UPPER_SNAKE_CASE` (例: `LOG_MESSAGE`)
- **名前空間名**: `lowercase_snake_case` (例: `audio_utils`, `ai_processing`)

2.3 メンバ変数

- **プライベートメンバ変数**: `m_` プレフィックスを使用 (例: `m_volumeLevel`)
- **スタティックメンバ変数**: `s_` プレフィックスを使用 (例: `s_instanceCount`)
- **ポインタ変数**: `p` プレフィックスは使用しない (例: `juce::AudioBuffer<float>* buffer`)

2.4 Boolean変数と関数

- **Boolean変数**: `is`, `has`, `can`, `should` などの接頭辞を使用 (例: `m_isInitialized`, `hasUndoHistory`)
- **Booleanを返す関数**: 同様に `is`, `has`, `can`, `should` などの接頭辞を使用 (例: `isPlaying()`, `canProcessBuffer()`)

2.5 JUCEコンポーネント

- **UIコンポーネント変数名**: 機能を表す名前の末尾にコンポーネント型を付加 (例: `playButton`, `volumeSlider`)

2.6 ファイル名

- **ヘッダーファイル**: `.h` または `.hpp` (プロジェクトで統一)
- **ソースファイル**: `.cpp`
- **ファイル名**: 対応するクラス名と同じ `PascalCase` (例: `AudioEngine.h`, `AudioEngine.cpp`)

---

3. フォーマットとスタイル

3.1 インデントとスペース

- **インデント**: 4スペースを使用（タブは使用しない）
- **行の長さ**: 120文字以内を推奨
- **括弧**: K&Rスタイル（開き括弧は行末、閉じ括弧は新しい行）
  `cpp if (condition) { // code } else { // code }`
- **スペース**: 演算子（`+`, `-`, `*`, `/`, `=`, `==`, `!=`, `&&`, `||` など）の前後にはスペースを入れる
- **関数呼び出し**: 関数名と開き括弧の間にスペースは入れない (例: `myFunction(arg1, arg2)`)
- **カンマ**: カンマの後にはスペースを入れる (例: `int a, b, c;`)

3.2 空行

- **論理的なブロック間**: 関連するコードブロックの間に空行を入れて可読性を高める
- **関数定義間**: 関数定義の間には1〜2行の空行を入れる
- **クラス定義内**: `public`, `protected`, `private` セクションの間に空行を入れる

3.3 ポインタと参照

- **アスタリスク(`*`)とアンパサンド(`&`)**: 型名の直後に配置 (例: `int* pointer;`, `const juce::String& text;`)

3.4 ヘッダーファイル

- **インクルードガード**: `#pragma once` を使用
- **インクルード順序**:
- 対応するソースファイルのヘッダー (例: `MyClass.cpp` なら `MyClass.h`)
- C++標準ライブラリヘッダー
- 外部ライブラリヘッダー (JUCE, Tracktion Engineなど)
- プロジェクト内ヘッダー
- 各グループ内ではアルファベット順にソート
- **前方宣言**: 可能な場合はインクルードの代わりに前方宣言を使用する

3.5 `const` の使用

- **`const` 正確性**: 変更されない変数やメンバ関数には `const` を積極的に使用する
- **メンバ関数の `const`**: オブジェクトの状態を変更しないメンバ関数は `const` 修飾する
- **引数の `const`**: 参照やポインタで渡される引数が関数内で変更されない場合は `const` 修飾する

3.6 `nullptr` の使用

- **ヌルポインタ**: `NULL` や `0` の代わりに `nullptr` を使用する

3.7 初期化

- **コンストラクタ初期化子リスト**: メンバ変数は可能な限り初期化子リストで初期化する
- **初期化順序**: 初期化子リストの順序はメンバ変数の宣言順序と一致させる

---

4. コメントとドキュメンテーション

4.1 コメントの目的

- **コードの意図を説明**: なぜそのように実装したのか
- **複雑なロジックの解説**: 理解が難しい部分の補足
- **前提条件や制約の明示**: コードが正しく動作するための条件

- **TODOやFIXME**: 将来の改善点や修正が必要な箇所

4.2 コメントスタイル

- **単一行コメント**: `//` を使用
- **複数行コメント**: `/* ... */` を使用（主にファイルヘッダーや大きなブロックのコメントアウト）
- **Doxygenスタイル**: APIドキュメンテーションにはDoxygen形式のコメントを使用（詳細はVibe Coding開発ガイド参照）

4.3 コメントすべき箇所

- **ファイルヘッダー**: ファイルの目的、作成者、更新日など
- **クラス、構造体、列挙型定義**: それぞれの役割や目的
- **関数、メソッド定義**: 機能、引数、戻り値、副作用、例外など
- **複雑なアルゴリズムやロジック**: 理解を助けるための説明
- **最適化やハック**: なぜそのようなコードが必要なのか
- **パブリックAPI**: 外部から使用されるインターフェースには詳細なドキュメントが必要

4.4 コメントすべきでない箇所

- **自明なコード**: コードを読めば明らか
- **古いコメント**: コードの変更に合わせてコメントも更新する
- **コードの単なる繰り返し**: コードが何をしているかを説明するのではなく、なぜそうしているかを説明する

4.5 TODOとFIXME

- `// TODO:`: 将来実装すべき機能や改善点
- `// FIXME:`: 既知の問題やバグで修正が必要な箇所
- コメントには担当者名や日付、関連するIssue番号を記載することを推奨

---

5. エラーハンドリング

5.1 基本方針

- **早期リターン**: エラーが発生したら速やかに処理を中断し、エラーを通知する
- **リソース解放**: エラー発生時にもリソース（メモリ、ファイルハンドル、ロックなど）が適切に解放されることを保証する (RAII)
- **明確なエラー情報**: エラーの原因や状況を特定できる情報を提供する

5.2 エラー通知方法

- **戻り値**: 単純な成功/失敗は `bool` で、より詳細な情報はエラーコードや `std::optional`, `std::expected` (C++23) を使用
- **例外**: 例外的状況やコンストラクタでのエラー通知に使用。リアルタイムオーディオ処理スレッドでは例外を避ける
- **JUCEのアサーション**: 開発中の論理エラー検出には `jassert` を使用
- **ログ**: エラーの詳細情報をログに出力する（詳細はVibe Coding開発ガイド参照）

5.3 例外安全

- **基本保証**: 例外が発生してもオブジェクトは有効な状態を保ち、リソースリークしない
- **強力な保証**: 例外が発生した場合、操作は失敗し、オブジェクトの状態は操作前の状態に戻る
- **例外を投げない保証**: 関数が例外を投げないことを `noexcept` で明示する（特にリアルタイム処理）

5.4 オーディオ処理におけるエラーハンドリング

- **リアルタイム制約**: オーディオコールバック内では、例外スロー、メモリ確保、ファイルI/O、ロックなどのブロッキング操作を避ける
- **エラーフラグ**: エラーが発生した場合は、アトミックなフラグを設定し、別のスレッドで処理する
- **サイレンス出力**: 回復不能なエラーが発生した場合、オーディオ出力はサイレンスにする

---

6. パフォーマンスと最適化

6.1 基本方針

- **測定ベースの最適化**: プロファイリングによってボトルネックを特定してから最適化を行う
- **可読性の維持**: 過度な最適化による可読性の低下を避ける
- **アルゴリズムの選択**: まずは適切なアルゴリズムとデータ構造を選択する

6.2 オーディオ処理のパフォーマンス

- **リアルタイム制約の遵守**: オーディオコールバック内での処理時間を最小限に抑える
- **メモリ確保の回避**: オーディオコールバック内での動的メモリ確保 (`new`, `delete`) を避ける。必要なバッファは事前に確保する

- **ロックフリープログラミング**: 可能な場合はロックフリーなデータ構造やアルゴリズムを使用する
- **SIMD命令の活用**: ループ処理やベクトル演算にはSIMD命令（SSE, AVXなど）の利用を検討する
- **インライン化**: パフォーマンスクリティカルな短い関数はインライン化を検討する

6.3 UIのパフォーマンス

- **描画処理の最適化**: `paint()` メソッド内での重い処理を避け、必要な部分のみ再描画する
- **非同期処理**: 時間のかかる処理（ファイル読み込み、ネットワーク通信など）はバックグラウンドスレッドで行い、UIの応答性を維持する
- **コンポーネントの効率的な配置**: `resized()` メソッド内でのレイアウト計算を効率的に行う

6.4 AI統合のパフォーマンス

- **非同期API呼び出し**: AIモデルへのリクエストは非同期で行い、UIやオーディオ処理をブロックしない
- **キャッシュ**: AIの応答結果をキャッシュし、同じリクエストに対する再計算を避ける
- **リクエストのバッチ処理**: 複数のリクエストをまとめて処理することで、API呼び出しのオーバーヘッドを削減する
- **ローカルモデルの活用**: Ghost Text機能など、リアルタイム性が重要な場合はローカルで動作する軽量モデルを検討する

---

7. メモリ管理

7.1 RAII (Resource Acquisition Is Initialization)

- **リソース管理**: メモリ、ファイルハンドル、ロックなどのリソースは、RAII原則に従ってクラスのコンストラクタで取得し、デストラクタで解放する
- **スマートポインタ**: 動的に確保されたメモリの管理には、`std::unique_ptr` や `std::shared_ptr` を使用する
- `std::unique_ptr`: 単一所有権。オブジェクトの所有権が明確な場合に使用
- `std::shared_ptr`: 共有所有権。複数のポインタが同じオブジェクトを指す可能性がある場合に使用。循環参照に注意
- **JUCEのスマートポインタ**: `juce::ScopedPointer`, `juce::OptionalScopedPointer`, `juce::ReferenceCountedObjectPtr` なども適切に使用

7.2 メモリリークの防止

- **`new` と `delete` のペア**: 手動で `new` した場合は必ず対応する `delete` (または `delete[]`) を行う。可能な限りスマートポインタを使用する
- **所有権の明確化**: オブジェクトの所有権を明確にし、二重解放や解放漏れを防ぐ
- **JUCEのリーク検出**: `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` マクロをクラス宣言に追加し、リーク検出を有効にする

7.3 メモリ確保のタイミング

- **リアルタイム処理中のメモリ確保禁止**: オーディオコールバック内など、リアルタイム性が要求される箇所での動的メモリ確保は避ける
- **事前確保**: 必要なメモリは初期化時や準備段階で事前に確保する
- **メモリプール**: 頻繁に確保・解放される小さなオブジェクトにはメモリプールを検討する

---

8. スレッドセーフティ

8.1 基本方針

- **共有データの保護**: 複数のスレッドからアクセスされる可能性のあるデータは、ミューテックス、クリティカルセクション、アトミック操作などで保護する
- **デッドロックの回避**: ロックの取得順序を統一するなど、デッドロックを発生させないように注意する
- **競合状態の防止**: データ競合が発生しないように、共有リソースへのアクセスを適切に同期する

8.2 JUCEの同期プリミティブ

- **`juce::CriticalSection`**: ミューテックス。基本的な排他制御に使用
- **`juce::ScopedLock`**: `juce::CriticalSection` と組み合わせてRAIIによるロック管理を行う
- **`juce::Atomic<T>`**: アトミック操作。単純な型の読み書きをアトミックに行う
- **`juce::WaitableEvent`**: スレッド間の同期や通知に使用
- **`juce::Thread`**: スレッドの作成と管理

8.3 オーディオ処理スレッドとUIスレッド

- **データ共有**: オーディオ処理スレッドとUIスレッド間でデータを共有する場合は、スレッドセーフな方法（アトミック変数、ロックフリーキュー、メッセージパッシングなど）を使用する
- **UI更新**: オーディオ処理スレッドから直接UIコンポーネントを更新しない。`juce::AsyncUpdater` やメッセージマネージャースレッド経由でUIを更新する

8.4 AI処理スレッド

- **非同期処理**: AIモデルへのリクエストは専用のバックグラウンドスレッドで行う
- **結果通知**: AI処理の結果は、スレッドセーフなキューやコールバックを通じてメインスレッドやUIスレッドに通知する

---

9. JUCE特有の規約

9.1 `juce::String` の使用

- **文字列操作**: `std::string` よりも `juce::String` を優先的に使用する（UTF-8対応、豊富なユーティリティ）

9.2 コンポーネントのライフサイクル

- **`paint()`**: 描画処理のみを行う。重い処理は避ける
- **`resized()`**: レイアウト処理のみを行う。子コンポーネントのサイズと位置を設定する
- **`setLookAndFeel()`**: LookAndFeelクラスを使用してUIの見た目をカスタマイズする

9.3 メッセージマネージャー

- **イベント処理**: UIイベントや非同期処理の結果通知は、JUCEメッセージマネージャーを通じて行う
- **`juce::Component::postCommandMessage()`**: 他のコンポーネントにコマンドを送信する

9.4 ValueTreeの使用

- **状態管理**: アプリケーションの状態やパラメータ管理には `juce::ValueTree` の使用を検討する（アンドゥ・リドゥ機能との連携が容易）

9.5 オーディオプラグイン開発

- `juce::AudioProcessor`: オーディオプラグインの基本クラス。パラメータ管理、状態保存、オーディオ処理などを実装
- `juce::AudioProcessorEditor`: プラグインのGUIエディタ

---

10. Tracktion Engine特有の規約

10.1 EngineとEdit

- `tracktion_engine::Engine`: アプリケーション全体で一つのインスタンスを保持
- `tracktion_engine::Edit`: プロジェクト（ソング）を表す。複数のEditを同時に開くことも可能

10.2 オブジェクトのライフサイクル

- **Tracktion Engineオブジェクト**: Tracktion Engineが管理するオブジェクト（Track, Clip, Pluginなど）は、直接 `delete` しない。EngineやEditのメソッドを通じて操作する
- `tracktion_engine::ObjectDeleter`: Engineがオブジェクトを非同期に削除するための仕組み

10.3 変更通知

- `tracktion_engine::ValueTreeAllEventListener`: Edit内の変更を監視するために使用
- `tracktion_engine::ChangeBroadcaster`: 特定のオブジェクトの変更を通知する

10.4 スレッドセーフティ

- **Engineのロック**: Tracktion Engineの内部状態にアクセスする際は、Engineのロック（`tracktion_engine::Engine::Lock l (engine);`）を取得する必要がある場合がある。ドキュメントを確認すること

10.5 拡張性

- **カスタムClipタイプ**: 独自のClipタイプを作成して機能を拡張可能
- **カスタムPluginタイプ**: 独自のPluginタイプ（内蔵エフェクトなど）を作成可能

---

11. AI統合コードの規約

11.1 モジュール性

- **疎結合**: AIクライアント、プロンプト生成、レスポンス解析などの機能を独立したモジュールとして設計する
- **インターフェース分離**: AIサービスプロバイダー（Claude, ChatGPT, Geminiなど）を切り替えられるように、共通のインターフェースを定義する

11.2 設定可能性

- **APIキー管理**: APIキーやエンドポイントURLは、設定ファイルや環境変数から読み込むようにし、コードにハードコーディングしない
- **モデル選択**: 使用するAIモデルは実行時に選択・変更できるようにする

11.3 エラーハンドリングとフォールバック

- **APIエラー処理**: ネットワークエラー、認証エラー、レート制限などのAPIエラーを適切に処理する
- **タイムアウト**: AIリクエストには適切なタイムアウトを設定する
- **フォールバック**: AIサービスが利用できない場合に備え、基本的なフォールバック機能（ローカルでの簡易生成など）を検討する

11.4 データ形式

- **プロンプト**: AIモデルへの入力プロンプトは、構造化された形式（JSONなど）で生成することを推奨
- **レスポンス**: AIモデルからの応答もJSONなどの構造化データとして解析する
- **MIDIデータ**: AIが生成するMIDIデータは、標準的なフォーマット（例: MIDIノート番号、開始時間、デュレーション、ベロシティ）で扱う

---

12. テストコードの規約

12.1 テストフレームワーク

- **Catch2**: 単体テストにはCatch2を使用する（Vibe Coding開発ガイド参照）

12.2 テストの原則 (FIRST)

- **Fast**: テストは高速に実行できること

- **Independent/Isolated**: 各テストは他のテストから独立しており、実行順序に依存しないこと
- **Repeatable**: 何度実行しても同じ結果が得られること（外部環境に依存しない）
- **Self-validating**: テスト自身が成功か失敗かを判断できること（手動での結果確認が不要）
- **Timely**: テストは本番コードと同時に（または先に）書かれること (TDD)

12.3 テストの種類

- **単体テスト**: クラスや関数の最小単位の機能をテストする
- **統合テスト**: 複数のモジュールが連携して正しく動作することをテストする
- **UIテスト**: GUIの操作や表示をテストする（JUCEのテストユーティリティや外部ツールを検討）

12.4 モックとスタブ

- **依存関係の分離**: テスト対象が外部モジュールに依存している場合、モックオブジェクトやスタブを使用して依存関係を分離する
- **Google Mock**: モックオブジェクトの作成にはGoogle Mockなどのライブラリの利用を検討

12.5 テストカバレッジ

- **目標カバレッジ**: 主要なモジュールについては高いテストカバレッジ（例: 80%以上）を目指す
- **カバレッジツール**: gcov, lcov, OpenCppCoverageなどのツールを使用してカバレッジを測定する

---

13. コードレビュー

13.1 レビューの目的

- **品質向上**: バグの早期発見、設計の改善
- **知識共有**: コードや設計に関する知識をチーム内で共有
- **規約遵守**: コーディングルールが守られていることを確認
- **学習機会**: レビューを通じて開発者個人のスキルアップを促進

13.2 レビュープロセス

- **プルリクエストベース**: GitHubなどのプルリクエスト機能を使用してレビューを行う
- **レビュアー**: 1名以上のレビュアーを割り当てる（可能であればシニア開発者を含む）

- **セルフレビュー**: プルリクエストを作成する前に、開発者自身がコードをレビューする
- **建設的なフィードバック**: 指摘は具体的かつ建設的に行い、人格攻撃は避ける
- **迅速な対応**: レビューコメントには迅速に対応し、議論を通じて合意形成を図る

13.3 レビュー観点

- **機能性**: 要件を満たしているか
- **設計**: SOLID原則、デザインパターンなどが適切に使用されているか
- **可読性**: コードは理解しやすいか
- **保守性**: 将来の変更や拡張が容易か
- **パフォーマンス**: ボトルネックや非効率な処理がないか
- **エラーハンドリング**: エラーケースが適切に処理されているか
- **テスト**: 十分なテストコードが書かれているか
- **ドキュメンテーション**: コメントやAPIドキュメントが適切か
- **規約遵守**: コーディングルールに従っているか

---

14. ツールと自動化

14.1 静的解析

- **Clang-Tidy**: コードの潜在的な問題やスタイル違反を検出するためにClang-Tidyを使用する
- **設定ファイル**: プロジェクトルートに `.clang-tidy` ファイルを配置し、チェック項目を設定する

14.2 フォーマッタ

- **Clang-Format**: コードのフォーマットを自動的に整形するためにClang-Formatを使用する
- **設定ファイル**: プロジェクトルートに `.clang-format` ファイルを配置し、フォーマットスタイルを設定する
- **IDE連携**: エディタやIDEの保存時自動フォーマット機能を有効にする

14.3 CI/CD

- **GitHub Actions**: ビルド、テスト、静的解析、フォーマットチェックを自動化するためにGitHub Actionsを使用する（詳細はVibe Coding開発ガイド参照）
- **自動デプロイ**: リリース時にはインストーラーのビルドと署名、リリース配布を自動化する

14.4 Cursor (AI支援)

- **コード生成**: CursorのAI機能を使用して、定型的なコードや複雑なロジックの初期実装を効率化する
- **リファクタリング**: AIによるリファクタリング提案を活用してコード品質を向上させる
- **バグ分析**: AIにコードを分析させ、潜在的なバグや改善点を特定する
- **ドキュメント生成**: コメントやREADMEの草案をAIに生成させる

注: このコーディングルールは、プロジェクトの進行やチームの成長に合わせて見直され、改訂されることがあります。変更提案は歓迎します。