④ 性能向上：Webアプリ 性能比較レポート（Sample）

1. 基本情報

- 受講者氏名：田中 太郎
- 作成日：2025年10月1日
- 対象工程：④ 性能向上（Webアプリ構造最適化・性能比較）
- 比較対象条件：
  - 条件A：基本設計（HTML5/CSS3/JavaScript ES6）
  - 条件B：モジュール最適化（ES6モジュール・CSS変数・コンポーネント化）
  - 条件C：バンドル最適化（Webpack・圧縮・チャンク分割）

---

2. 実験概要

- 実験目的：Webアプリ構造最適化による性能向上とコード品質改善の検証
- 比較タイプ：**リファクタリング・パフォーマンス最適化**（同一アプリケーションのコード構造改善とビルドプロセス最適化による性能比較）
- 実装基盤：単一HTMLファイルからES6モジュール構造への移行
- 比較観点：
  - 性能指標（成功率、正確性、処理時間）
  - 構造品質指標（保守性、可読性、拡張性、初期ロード時間、バンドルサイズ、Lint指摘件数、依存循環有無）
- 使用データセット：経費管理システムのテストデータ（100件の経費登録処理）
- 実行環境（OS / CPU / メモリ / ブラウザ / ツール / バージョン）：Windows 11 / Core i7-8750H / 16GB / Chrome 130 / VS Code 1.94 / Node.js 18.17
- 実行回数：各条件10回実行して平均値を算出
- 計測方法：
  - 性能：代表操作の平均処理時間（DevTools Performance、User Timing API）
  - 初期ロード：Lighthouse（FCP/TTI）、WebPageTest
  - バンドルサイズ：ビルド成果物合計（KB）、webpack-bundle-analyzer
  - 保守性/可読性：ESLint指摘件数、SonarQube、Code Climate
  - 拡張性：新機能追加時の改修行数・影響モジュール数（新しい経費カテゴリ追加で測定）
  - 依存循環：dependency-cruiser、madge
- 許容基準値：FCP < 1.5秒、TTI < 3秒、ESLint指摘 < 10件、依存循環 = 0件

---

3. 実験結果

3.1 性能評価

比較条件	成功率（%）	正確性（%）	処理時間（秒）	備考
条件A	95.0	98.5	0.85	基本実装、DOM操作多数
条件B	98.5	99.2	0.52	モジュール化によるコード整理効果
条件C	99.0	99.5	0.31	バンドル最適化とコード分割効果

3.2 構造品質評価

指標	条件A	条件B	条件C	測定方法/備考
初期ロード FCP（秒）	2.1	1.4	0.9	Lighthouse, 5回測定平均
初期ロード TTI（秒）	4.2	2.8	1.6	Lighthouse, 5回測定平均
バンドル合計（KB）	285	198	95	ビルド成果物合計（gzip圧縮後）
ESLint指摘（重大/警告）	28/45	3/12	0/5	@typescript-eslint/recommended
コードスメル（件）	15	4	1	SonarQube分析結果
重複コード率（%）	12.3	3.8	1.2	SonarQube分析結果
モジュール数（個）	1	8	12	src配下のJS/TSファイル数
依存循環（件）	3	0	0	dependency-cruiser検証
新機能追加の改修行数（diff行）	127	43	28	新経費カテゴリ追加時
影響モジュール数（個）	1	3	2	同上（コンポーネント分離効果）

---

4. 考察

- 性能差の要因：
  - 条件Aは単一ファイルによる肥大化とDOM操作の非効率性が影響
  - 条件Bではモジュール分離により責務が明確化され、パフォーマンス向上
  - 条件Cではcode splitting、tree shaking、圧縮により大幅な軽量化を実現
- 構造品質への影響：
  - モジュール化により保守性が劇的に向上（ESLint指摘数73%削減）
  - コンポーネント設計により重複コード率を90%削減
  - 依存関係の明確化により循環依存を完全解消
- 改善が見られた点：
  - 初期ロード時間57%短縮（FCP: 2.1秒→0.9秒）
  - バンドルサイズ67%削減（285KB→95KB）
  - 新機能追加時の改修コスト78%削減
- 劣化が見られた点：
  - モジュール数増加による複雑性（管理コスト増）
  - ビルドプロセスの導入（開発環境の設定コスト）
- 想定外の結果や原因：
  - 条件Bでも十分な性能向上効果（予想以上のモジュール化効果）
  - バンドル最適化による劇的なサイズ削減効果

---

5. 結論

- 最適条件の選定理由： 条件C（バンドル最適化）を最適とする。理由は全指標で最高性能を達成し、特に初期ロード時間、バンドルサイズ、保守性において顕著な改善を示したため。開発コストの増加はあるものの、長期的な保守性向上による総コスト削減効果が大きい。
- 今後の改善案：
  - Progressive Web App（PWA）対応によるオフライン機能追加
  - WebAssembly活用による計算処理の高速化
  - Service Worker実装によるキャッシュ戦略最適化
  - Critical CSS inline化による更なる初期ロード高速化
- 他工程への適用可能性： モジュール設計原則とバンドル最適化手法は他のWebアプリ開発工程でも適用可能。特に③経費登録ワークフローのフロントエンド化、⑤電力需要予測のダッシュボード開発に応用可能。

---

6. 再現手順

1. 実験用データセットの準備：
   - 経費データ100件のJSON形式テストデータ作成
   - Lighthouse CI設定ファイル準備
   - パフォーマンス計測用のPuppeteerスクリプト作成
2. 条件A/B/Cの環境構築：
   - 条件A: 基本HTML/CSS/JS構成
   - 条件B: ES6モジュール、Babel、ESLint導入
   - 条件C: Webpack、TerserPlugin、MiniCssExtractPlugin設定
3. 各条件での性能・構造品質測定：
   - `npm run lighthouse` でLighthouse測定
   - `npm run bundle-analyzer` でバンドル分析
   - `npm run lint` でコード品質測定
4. 結果集計と分析：
   - 測定結果をCSV形式で集計
   - 統計分析とグラフ可視化

---

7. 添付・参考資料

- 添付ファイル名：
  - performance_comparison_results.xlsx（詳細測定データ）
  - lighthouse_reports/（Lighthouseレポート）
  - bundle_analysis.html（バンドル分析結果）
  - code_quality_metrics.json（コード品質指標）
- 参考URLや文献：
  - Web Performance Optimization: https://web.dev/performance/
  - Webpack Bundle Optimization: https://webpack.js.org/guides/optimization/
  - ESLint Configuration Guide: https://eslint.org/docs/user-guide/configuring
  - サイバーパンクUI設計パターン: 内部スタイルガイド参照