

測定器データ変換ロジック開発ガイドライン

1. はじめに

1.1 背景と目的

RAMPプラットフォームでは、多様な測定器メーカーから送られてくる観測データを収集・統合し、共通のフォーマット（Standard RAMIS Format）で蓄積・活用することを目指しています。しかし、測定器ごとのデータフォーマット（CSVの列構成やヘッダ有無など）は多岐にわたり、これらを一つ一つプラットフォーム側で吸収・変換することは大きな開発コストとなっています。

本ガイドラインは、測定器メーカー様（または変換ロジックを担当するパートナー様）向けに、「**測定器独自のCSVフォーマット**」を「**RAMP共通JSONフォーマット**」に変換するプログラム（クラス）を開発いただくための技術資料です。測定器の仕様を最も理解している皆様に変換ロジックを作成いただくことで、データ品質の担保とスムーズなプラットフォーム収容を実現することを目的としています。

1.2 期待する成果物

- C# クラスファイル (.cs) 1つ以上
 - 後述する基底クラス `JsonCreatorBase` を継承し、貴社測定器のフォーマット変換処理を実装したもの。

2. アーキテクチャ概説（初心者向け：Nintendo Switchに例えると？）

本システムでは、拡張性とメンテナンス性を高めるために、**Factoryパターン** と **Strategyパターン** という設計手法（デザインパターン）を採用しています。この仕組みは、「**Nintendo Switch**」に例えると非常に分かりやすくなります。

- **プラットフォーム (RAMP)** = 「Switch本体」
- **Factory (工場)** = 「ゲームカード切り替え機（セクター）」
- **Strategy (変換ロジック)** = 「ゲームソフト（カセット/カード）」

2.1 Strategyパターン：ソフト交換式

Switch本体（RAMP）は、ゲームソフトの具体的な中身を知りません。しかし、ソフトが**「決められた形状（規格）」**で作られてさえいれば、それを差し込むだけでどんなゲームでも動かすことができます。

- **共通の枠組み (JsonCreatorBase):** ゲームソフトの**「端子の形状」や「サイズ」**の規格です。この規格に合わないソフトは挿入できません。
- **具体的な実装 (貴社作成クラス):** この規格に合わせて作られた**「個別のゲームソフト」**です。中身（変換ルール）は自由ですが、外側（入力と出力の形式）は規格通りに作る必要があります。

```
// 【Strategyの構造】
// 1. 共通の規格（仮想メソッドとして定義）
public class JsonCreatorBase {
    public virtual RequestJson create(List<String> data) {
```

```
        throw new NotImplementedException(); // 実装忘れ防止のため
    }
}

// 2. 具体的な実装（オーバーライド）
public class MyMonitor : JsonCreatorBase {
    public override RequestJson create(List<String> data) {
        // ... ここに実際の変換コードを書く ...
    }
}
```

2.2 Factoryパターン：生成ロジックの分離（ソフト切り替え機）

このパターンの本質は、「**どれを動かすか（選定・切替）**」と「**どう動かすか（実行）**」を切り離すことにあります。Factory（ソフト切り替え機）は、入力されたファイル名を見て「今回はこのソフトを使う」と**決定・切り替え**を行います。

Switch本体（RAMP）は、「具体的にマリオなのかゼルダなのか」を意識しません。「切り替え機が選んでくれたソフトを、とりあえず動かす」ことだけに集中します。**その代わり**、「ファイル名を見て、適切な変換クラス（ソフト）を選び出し、セットする」役割は、すべてこのFactory機能に一任されています。

皆様のお役割とお願い: Factory クラス側への個別判定ロジックの組み込みは、すべてプラットフォーム側（発注側）で行います。皆様をお願いしたいのは以下の2点のみです。

1. **カセットの作成:** 規格通りの単独で動作する変換クラス（.cs ファイル）を作成すること。
2. **【重要】識別ルール（ファイル命名規則）の遵守:**
 - 本システムは、ファイル名に含まれる特定の文字列（型番など）で処理クラスを切り替えます。
 - 貴社向けの識別ルールは以下の通り決定しますので、送信するファイル名には必ずこの文字列を含めてください。

【識別ルール（プレースホルダ）】

- **対象文字列:** [ここにRAMP側で決定した識別用文字列を記載します（例：SAGA_AIR）]
- **ルール:** ファイル名のどこかに上記の文字列が含まれていること。

3. 実装ガイド

3.1 開発環境

- 言語: C# (.NET Core / Standard)
- 提供ライブラリ: RampTelemeterCmn.dll (共通型定義が含まれます)

3.2 実装手順

1. 基底クラス **JsonCreatorBase** を継承する

```

using TransportableCollect.JsonCreator;

// クラス名は測定器名などが分かりやすいです（例：MyInstrumentV1）
public class MyInstrumentV1 : JsonCreatorBase
{
    // コンストラクタで都道府県コード（_prefNumber）を設定
    public MyInstrumentV1() { _prefNumber = 99; }

    // createメソッドをオーバーライド（上書き）して変換ロジックを書く
    public override RequestJson create(List<String> oneLineList)
    {
        // ここにロジックを記述
    }
}

```

2. create メソッドを実装する

- 入力: `List<String> oneLineList` (CSVファイルの各行が文字列として格納されたリスト)
- 出力: `RequestJson` (共通フォーマットのデータ保持クラス)

実装イメージ（ヘルパーメソッドの配置）：

```

public override RequestJson create(List<String> oneLineList)
{
    RequestJson ret = new RequestJson();

    // 1行目を取得（ヘッダなしの場合）
    String line = oneLineList[0];
    String[] columns = line.Split(','); // カンマ区切りなどで分解

    // Mp（モニタリングポスト）オブジェクトの作成
    Mp oneMp = new Mp();
    oneMp.MPNumber = int.Parse(columns[0]); // 例：1列目がMP番号

    // 測定データオブジェクトの作成
    Mp.Measuredata data = new Mp.Measuredata();

    // 日時変換（基底クラスの便利メソッド convertDatetime も利用可能）
    data.MeasurementDate = convertDatetime(columns[1], "yyyy/MM/dd
HH:mm:ss");

    // 線量率などの測定値をセット
    List<Mp.Measure> measures = new List<Mp.Measure>();

    // 以下のようなローカルヘルパーメソッド（独自の例外処理等を追加）を利用して値を安全に格納
    // することを推奨します。
    addOneMeasure(measures, MeasureNumber.DOSE, columns[2], 60, "空間線量
率");

    // 構造体にセットしてリターン
    data.Measure = measures.ToArray();
}

```

```
oneMp.MeasureData = new Mp.Measuredata[] { data };
ret.MPs = new Mp[] { oneMp };

return ret;
}

// クラス内に記述する値設定用ヘルパーメソッドの例
private void addOneMeasure(List<Mp.Measure> measureList, int measureNumber,
String data, int interval, String measureName)
{
    if (String.IsNullOrEmpty(data) == false)
    {
        // TODO: ここで例外を捕捉し、システムが停止しないように必ず防御的プログラミングを行ってく
        ださい。

        Mp.Measure ret = new Mp.Measure();
        ret.MeasureNumber = measureNumber.ToString();
        ret.MeasuredValue[0] = Decimal.Parse(data); // 実際にはTryParseを推奨
        ret.MeasureInterval = interval;
        ret.MeasureName = measureName;
        measureList.Add(ret);
    }
}
```

4. 品質の担保（型安全性と検証）

CSVなどのテキストデータは、想定外の値（空文字、不正な文字、欠損）が含まれることがあり、システムエラーの温床となります。本プラットフォームでは「コンパイル時の型チェック」と「実行時のバリデーション」によって品質を担保します。

4.1 静的型付けによる保証 (RequestJson クラス)

変換先の RequestJson クラスは、C# の強力な型システムによって定義されています。例えば、MPNumber は int (整数) 型、MeasuredValue は decimal (数値) 型として定義されています。これにより、「誤って文字列を数値項目に代入する」といった単純なミスは、プログラムをビルド（コンパイル）した時点でエラーとして検出できるため、不具合の混入を未然に防げます。

4.2 実装時の注意点（バリデーションの徹底）

プラットフォームが提供するサンプルコード等は、正常系動作の確認を目的としたものであり、例外ハンドリングを一部省略している場合があります。しかし、皆様が開発される変換用カセットにおいては、必ず以下の防御的プログラミングを行って、堅牢な実装としていただきますようお願いいたします。

- 数値変換時のチェック:** int.Parse や decimal.Parse をいきなり呼ぶのではなく、空文字チェックや TryParse を活用してください。
- 必須項目の確認:** 測定日時 (MeasurementDate) や MP番号 (MPNumber) など、システム上必須となる項目が欠損している場合は、エラーログを出力してその行をスキップするか、例外 (Exception) をスローして処理を中断するような設計にしてください。
- 例外処理:** try-catch ブロックで囲み、予期せぬフォーマット異常（1行内のカラム数不足など）が起きてもシステム全体を停止させず、エラー原因が特定できるようにしてください。

5. 実装ルール (Do's and Don'ts)

品質と保守性を担保するため、以下のルールを遵守してください。

5.1 必須事項 (Must)

- **継承とコンストラクタ**: 必ず `JsonCreatorBase` を継承し、コンストラクタで都道府県コード (`_prefNumber`) を設定すること。
- **必須項目の設定**: `RequestJson` 内の以下の項目は必ず値を設定すること。
 - `MPs.MPNumber`: モニタリングポスト番号
 - `MPs.MeasureData.MeasurementDate`: 測定日時 (yyyyMMddHHmmss形式文字列)
 - `MPs.MeasureData.DeviceNumber`: 測定器番号
- **座標変換**: 緯度経度が「度分秒 (DMS)」形式で来る場合は、必ず `convertLatitudeFrom60To10 / convertLongitudeFrom60To10` を使用して「10進数 (Decimal)」に変換すること。
- **命名規則**: クラス名は `[都道府県名][測定器名].cs` (例: `SagaAirMonitor.cs`) とし、ファイル名とクラス名を一致させること。

5.2 禁止事項 (Don't)

- **例外の握りつぶし**: `try-catch` ブロックで例外を捕捉した後、何もせずに処理を続行しないこと。パースエラーが発生した行は、エラーログを出力してスキップするか、致命的な場合は処理を中断すること。
- **マジックナンバーの多用**: 測定項目番号などは、可能な限り `MeasureNumber` クラス等の定数定義を使用すること (定数定義ファイルは別途貸与します)。
- **外部通信**: 変換ロジック内で外部APIを呼び出したり、ファイルシステムへの書き込みを行ったりしないこと (純粋なデータ変換のみを行うこと)。

6. 静的解析と自動チェックについて

現在、本プロジェクトでは Roslyn Analyzer 等による**自動的なコーディング規約チェック**は導入されていません。したがって、上記の「必須事項」「禁止事項」はビルドエラーにはなりませんが、**コードレビュー時の必須チェック項目**となります。

6.1 コンパイラによる強制と限界

C#の言語仕様上、以下の挙動となります。

- **継承の強制**: 工場クラス (`JsonCreatorFactory`) が `JsonCreatorBase` 型を返却するため、継承していないクラスを工場に登録しようとすると**コンパイルエラー**になります (これは確実に守られます)。
- **メソッド実装の強制**: 基底クラスの `create` メソッドは `virtual` (仮想メソッド) で定義され、デフォルトで例外 (`NotImplementedException`) をスローする機能になっています。そのため、皆様のクラスでメソッドのオーバーライド (上書き) を忘れても**コンパイルは通ってしまい、実行時にエラー (クラッシュ) ****となります。
 - 注意: 作成するクラスでは必ず `override` キーワードを用いて `create` メソッドを実装してください。

7. 作業量の見積もりについて

本作業の工数を見積もる際は、以下の要素を考慮してください。

7.1 作業単位 (Units)

見積もりの基本単位は「**1フォーマット (1クラスファイル)**」です。

- 同じ測定器でも、ファームウェアバージョンによって出力CSVの列順が変わる場合は、原則として別クラス（別タスク）としてカウントします。

7.2 難易度係数

1クラスあたりの作業時間は、以下の要素により変動します。

- **基本 (x1.0)**: 単純なCSV（ヘッダなし、固定列、単純マッピング）。
- **複雑 (x1.5)**: ヘッダ解析が必要、あるいはデータ行が複数行にまたがる場合。
- **高度 (x2.0~)**: バイナリデータのデコードが必要、あるいは複雑な計算ロジック（単位変換や補正計算）を含む場合。

7.3 想定工数（目安）

- **仕様把握・設計**: 0.5日 / フォーマット
- **実装 (コーディング)**: 0.5日～1.0日 / フォーマット
- **単体テスト (正常系・異常系)**: 0.5日～1.0日 / フォーマット

合計: 1フォーマットあたり、**1.5日～3.0日 (12H～24H)** 程度が標準的な目安となります。 ※既定のテンプレート ([SagaTransportable.cs](#) 等) を流用できる場合は、さらに短縮可能です。 ※座標変換ロジックや日付変換ロジックは基底クラスで提供されているため、これらの再実装工数は不要です。