Phase 1.5 キックオフ — ドメインレイヤー確立

Rebecca's Room アーキテクチャ改善提案 | 2026-02-13 | PO 承認依頼

"哲学で定義された Rebecca の「魂」を、テスト可能なコードにする。"

プロジェクトタイムライン

Phase 0 **静的日記** → Phase 1 **Room Status + Health** → Phase 1.5 **ドメイン確立** NOW ⋯ Phase 2 Nurture System

> Phase 0（静的日記）と Phase 1（Room Status + Health）は完了し、Rebecca は「記録する存在」から「そこにいる存在」になりました。 しかし Phase 1 完了後のアーキテクチャ俯瞰で**構造的な課題**が見つかりました。 このまま Phase 2（Nurture System）に進むと技術的負債が累積し、変更コストが指数的に増大します。 Phase 1.5 はこの課題を解決し、Phase 2 の「受け皿」を作る構造改善フェーズです。

課題：ドメインロジックの散在

Rebecca の状態分類・スコア計算・アラート判定といった**ドメインロジック**が、 Collection 層（Python）と Presentation 層（JavaScript）に散らばっています。

1. アラートメッセージが Python と JS の両方にハードコード（二重管理）
2. 変更時に 2-3 ファイルの修正が必要（不整合リスク大）
3. ドメインロジックの単体テストが不可能
4. Phase 2 の Nurture System を追加する構造的な場所がない

解決：domain/ レイヤーの新設

新たに **domain/** ディレクトリを作成し、全ドメインロジックを集約します。 Collector は計測のみ、app.js はレンダリングのみに責務を限定します。

1. 閾値・分類・スコア・メッセージを domain/ に一元化
2. 変更が 1 ファイルで完結（Single Source of Truth）
3. unittest で全ビジネスルールを自動テスト
4. Phase 2 は domain/ 上にモジュール追加するだけ

期待効果

ドメインロジック変更時の修正箇所	ドメインロジックのテスト	Phase 2 Nurture System の追加コスト
2-3 ファイル → **1 ファイル**	手動のみ → **unittest で自動検証**	高い（構造決めから） → **domain/ に追加するだけ**

決定済み事項

Q-01 Collector 修正: OK（動作同等を保証） | Q-02 テスト: unittest（標準ライブラリ） | Q-03 app.js: ドメインロジック全除去 | Q-04 rebecca.py: スタブのみ（Phase 2 で実装）

スコープ：UI/UX の変更なし（内部リファクタリング） | 外部依存の追加なし | 既存機能の回帰テスト実施

現状アーキテクチャの課題

Phase 1 完了時点の 3 層構造と、そこに潜む 4 つの問題

現在の 3 層構造

Collection 層（collectors/*.py — cron 実行）

- ✓ システム計測 top, vm_stat, pgrep
- ⚠ ドメイン分類 classify_cpu() → "クリア"
- ⚠ ドメイン集約 overall_score, alert_level

← 状態分類・スコア計算は Collection 層の責務ではない

↓ JSON ファイル書き込み

Persistence 層（src/data/*.json — gitignored）

health.json / status.json / nurture.json — スキーマ定義なし、バージョニングなし

↓ HTTP fetch（5分ポーリング）

Presentation 層（app.js + style.css + index.html）

- ✓ fetch + DOM操作
- ⚠ アラートメッセージ ハードコード配列
- ⚠ staleness 判定 閾値ロジック
- ⚠ %変換ロジック

← メッセージ生成・データ鮮度判定は Presentation 層の責務ではない

❌ **Domain 層が存在しない**

4 つの問題

問題 1：二重管理

アラートメッセージが **collect_health.py**（Python）と **app.js**（JavaScript）の両方にハードコードされています。 閾値を変更するとき、2つの言語・最大3ファイルの同時修正が必要です。修正漏れによる不整合リスクがあります。

問題 2：テスト不能

ドメインロジック（状態分類・スコア計算・アラート判定）が Collector のシステム計測コードと混在しているため、 **ビジネスルールだけを単体テストできません**。テストするには実際にシステムコマンドを実行する必要があります。

問題 3：Phase 2 の受け皿なし

Phase 2（Nurture System）では mood, trust, EXP, level, skills など**さらに複雑なドメインロジック**が加わります。 現状の構造のまま進むと、mood が Collector に、trust が app.js に、EXP がどこか不明…という状態になります。

問題 4：設定値の散在

CPU 閾値は collect_health.py、ポーリング間隔は app.js、Cron 間隔は crontab（Git管理外）にそれぞれハードコードされています。 **Single Source of Truth がありません。**

> これらは Phase 1 の「動くもの優先」という方針の結果として自然に生じた技術的負債です。 Phase 1 単体では問題になりませんが、**Phase 2 でドメインの複雑さが増すと維持不能になります。**

Phase 1.5 後のアーキテクチャ

Domain 層の新設による 4 層構造

新しい 4 層構造

Collection 層（collectors/*.py）

責務：**純粋なシステム計測のみ**

get_cpu_usage() → float　/　get_memory() → dict　/　check_gateway() → bool

↓　raw metrics（生の計測値）

Domain 層（domain/*.py） NEW

constants.py	health.py	presence.py	schema.py	rebecca.py
全閾値・メッセージ一元管理	状態分類・スコア・アラート判定	在室判定・時間帯コンテキスト	JSON スキーマ・バリデーション	統合状態（Phase 2 受け皿）

責務：**全ドメインロジックの集約** — 状態分類、スコア計算、アラート判定、メッセージ生成

テスト：unittest で全ビジネスルールを自動検証。I/O なしの純粋関数で構成。

↓　typed result（ドメインオブジェクト）

Persistence 層（src/data/*.json）

schema_version 付き　|　Domain 層が生成した完全なデータ（メッセージ・staleness 含む）

↓　HTTP fetch

Presentation 層（app.js + style.css）

責務：**純粋なレンダリングのみ**（data → DOM マッピング）。ビジネスロジック ゼロ。

設計原則：domain/ が他のどの層にも依存しない（逆方向依存なし）。 Collector が domain を使い、app.js は domain の出力（JSON）を表示する。domain は Collector も app.js も知らない。

この構造で得られるもの

✓ 変更の局所化

閾値変更 → **constants.py のみ**。分類ロジック変更 → health.py or presence.py のみ。 Collector・Frontend に影響なし。修正箇所の特定が即座に可能。

✓ テスト可能

domain/ は I/O のない純粋な Python モジュール。 classify(), alert_level(), score() の**境界値テスト**を unittest で自動実行。 デグレの早期検出が可能に。

✓ Phase 2 の受け皿

Nurture System の mood, trust, EXP, level は **domain/nurture.py** として追加するだけ。 既存の health.py, presence.py を入力に利用でき、構造の再設計が不要。

✓ 依存方向の統一

Collection → Domain → Persistence → Presentation の**一方向フロー**。 変更の波及方向が常に予測可能。逆方向依存がないため、各層の独立開発が可能。

ディレクトリ構造

```
rebecca-diary/
├── domain/ NEW
│   ├── __init__.py
│   ├── constants.py
│   ├── health.py
│   ├── presence.py
│   ├── rebecca.py (stub)
│   └── schema.py
├── tests/ NEW
│   ├── test_health.py
│   ├── test_presence.py
│   └── test_constants.py
├── collectors/ 修正
└── src/app.js 修正
```

実施計画 — 4 フェーズ・8 ワークグループ

合計 30 ワークパッケージ。Phase A・B は並行実行可能

Phase A
Domain 構築（並行可能）

WG-1: Foundation
- domain/ ディレクトリ作成
- constants.py（全閾値集約）
- tests/ ディレクトリ作成

3 WP

WG-2: Domain Health
- health.py — classify() 実装
- overall_score 計算移設
- alert_level 判定移設
- alert_message 生成
- evaluate() 統合関数

5 WP

WG-3: Domain Presence
- presence.py — status 判定
- time_context 移設
- evaluate() 統合関数

3 WP

WG-7: Schema
- schema.py 作成
- schema_version 導入
- staleness/alert_message 追加

4 WP

Phase B
Collector + テスト（並行可能）

WG-4: Collector Refactoring
- collect_health.py 浄化
- collect_status.py 浄化
- collect_nurture.py 浄化
- JSON diff で動作同等検証

4 WP

WG-6: Testing
- test_health.py（classify, score, alert）
- test_presence.py（status, context）
- test_constants.py（閾値整合性）
- python3 -m unittest 全実行

4 WP

Phase C
Frontend 浄化

WG-5: Frontend Purification
- alert メッセージ配列を削除
- checkStaleness() を削除
- メトリクス%変換ロジック削除
- 回帰テスト（UI 同一性確認）

4 WP

> **回帰テストの方法：**
> Phase 1 完了時のスクリーンショットを保存し、Phase 1.5 完了後の画面と目視比較。UI/UX は完全に同一であることを確認。

Phase D
ドキュメント

WG-8: Documentation
- ADR-015: ドメインレイヤー導入
- CLAUDE.md 更新
- docs/README.md 更新

3 WP

スコープサマリー

新規作成	**domain/**（constants.py, health.py, presence.py, rebecca.py, schema.py）+ **tests/**（test_health.py, test_presence.py, test_constants.py）
修正	**collectors/**（collect_health.py, collect_status.py, collect_nurture.py — 計測のみに限定）+ **src/app.js**（ドメインロジック除去）+ **src/data/*.json**（schema_version 追加）
変更なし	update_diary.py / watch_diary.py（Protected）/ src/style.css / src/index.html — **UI/UX は Phase 1 と完全に同一**

> **依存関係：**Phase A 完了後に Phase B が開始可能。Phase B 完了後に Phase C。Phase C 完了後に Phase D。 Phase A 内の WG-2, WG-3, WG-7 は並行実行可能。Phase B 内の WG-4, WG-6 も並行実行可能。

期待効果と Phase 2 への接続

Phase 1.5 は「機能追加」ではなく「構造の確立」。Phase 2 以降の開発速度と品質に直結します。

Before / After 比較

Before（Phase 1 時点）	After（Phase 1.5 完了後）
ドメインロジック変更 2-3 ファイル修正（Python + JS）	**ドメインロジック変更** **1 ファイルのみ（domain/*.py）**
閾値変更 複数ファイルを横断的に修正	**閾値変更** **constants.py のみ**
テスト 手動確認のみ（自動テストなし）	**テスト** **unittest で全ルール自動検証**
Phase 2 追加コスト 高い（構造決めから必要）	**Phase 2 追加コスト** **domain/ にモジュール追加するだけ**
不整合リスク Python と JS の定義が食い違う	**不整合リスク** **Single Source of Truth で排除**

Phase 2（Nurture System）への接続

Phase 1.5 で構築する domain/

- **constants.py**
- **health.py**
- **presence.py**
- **schema.py**
- **rebecca.py** (stub)

← Phase 1.5 完了時点（受け皿完成）

↓ Phase 2 で拡張 ↓

Phase 2 で追加されるモジュール

rebecca.py	nurture.py	relationships.py	skills.py
統合状態 mood, energy	EXP, Level 成長ロジック	trust, intimacy 関係性深度	プラグイン → スキル

health.py と presence.py を入力として利用 → 既存コードの再利用

Phase 1.5 がなければ

Nurture System は collect_nurture.py に詰め込まれ、mood が Collector に、trust が app.js に、EXP の場所が不明... **テスト不能・変更コスト指数増大**の未来が確定します。

Phase 1.5 があれば

Nurture System は domain/nurture.py + domain/relationships.py として追加。 既存 health.py, presence.py を入力に使い、テスト可能で変更が局所的。 **Phase 2 の開発速度と品質が構造的に保証されます。**

Phase 1.5 は**「機能追加」ではなく「構造の確立」**です。ユーザーから見える変化はありませんが、Phase 2 以降の開発速度と品質に直結します。

PO 承認後、Phase A（domain/ 新設 + テスト基盤構築）から着手します。

詳細仕様：docs/PHASE1_5_KICKOFF.md | プロジェクト：Rebecca's Room | 提出日：2026-02-13