派生開発推進協議会
AFFORDD

AFFORDDパートⅠ

生成AI時代の要求仕様と派生開発

斎藤 賢一
派生開発推進協議会 代表
エクスモーション エグゼクティブコンサルタント

仕様記述法は、20以上が乱立している

派生開発推進協議会
AFFORDD

自然文・要求記述
意図と理由を伝える
- **USDM**
- ユーザーストーリー
- ユースケース記述

振る舞い記述 (BDD / ATDD)
テストと直結
- Gherkin
- EARS
- Spec by Example

形式仕様 (数学ベース)
数学的厳密性
- Z notation
- Alloy
- TLA+
- B-method

モデルベース (図・表)
構造を視覚化
- UML / SysML
- 状態遷移表
- デシジョンテーブル

ドメイン特化
分野最適化
- OpenAPI / Protobuf
- AADL / SCADE
- OWL / SHACL

20以上の記法が乱立する中、なぜ私たちAFFORDDは USDM を選ぶのか？

USDM とは何か

派生開発推進協議会
AFFORDD

USDM = Universal Specification Describing Manner

（要求仕様を記述するための汎用手法。日本発、清水吉男氏が考案）

① 要求を「階層」で書く

要求を振る舞いとして記述し、上位の要求の範囲を狭めて下位の要求へ分解。

粒度をそろえて、全体像と細部を

同じ文書で両立できる

② すべてに「理由」を添える

なぜその要求が必要かを必ず明記。

読み手（人もAIも）の解釈ブレ、勘違いを防ぎ、仕様のひらめきを誘発。

必要なら仕様にも理由を

③ 要求から「仕様」を導く

要求記述の「目的語＊動詞」から、検証できる具体的な

（仕様）へ落とし込む。

要求↔仕様が対応づく＝テスト直結

一言でいうと…

「要求」＋「その理由」＋「具体的な仕様」を、階層構造でワンセットにして書く記法

だから —— 人にもAIにもテストにも、そのまま渡せる

USDM の構造 ― 実例で見る

派生開発推進協議会
AFFORDD

USDM の7つの構成要素

- **グループ** … 関連する要求、仕様のまとまり
- **キーワード（ラベル）** … 要求を識別する見出し
- **要求** … 実現したいこと（～できること）
- **理由** … なぜその要求が必要か
- **説明** … 要求の前提・補足
- **仕様** … 要求を具体化した振る舞い
- **仕様ラベル** … 個々の仕様の識別番号

記述例：キッチンタイマー

〔要求 KT01〕システムは、スタート操作で設定時間からカウントダウンを開始して残り時間を減算し、残り時間がなくなったらアラームを吹鳴する

【理由】設定した時間になったことに気づかないことを無くすため
【説明】待機状態でのみ開始でき、計時中は時間設定を受け付けない

＜スタート操作の受付＞ ← **仕様グループ**

・□□□**KT01-01** 「スタート」ボタン押下の判断はIOポートXXXXで判断する ← **仕様**

＜カウントダウンの開始＞

・□□□**KT01-02** 設定時間が0より大きいとき、設定時間を残り時間に複写して開始する

＜残り時間の減算＞

・□□□**KT01-03** カウントダウン中、1秒ごとに残り時間を1秒減算する

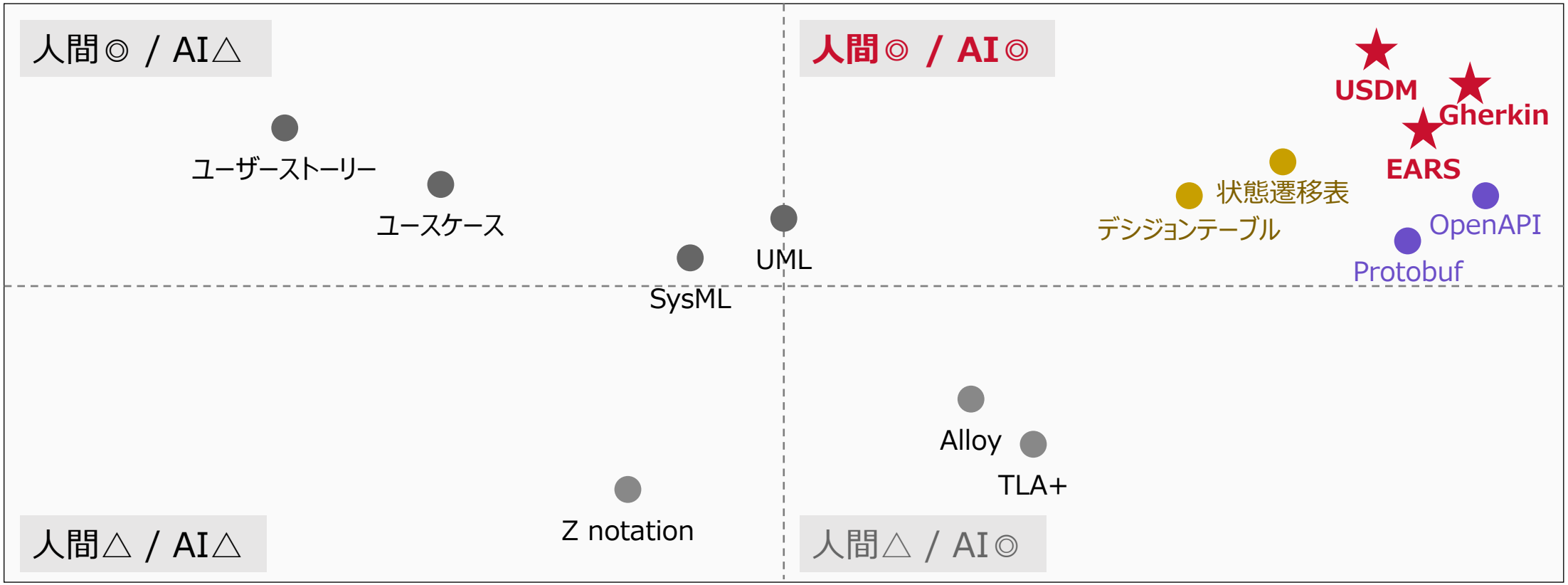
＜アラームの吹鳴＞

・□□□**KT01-04** 残り時間が0になったらカウントダウンを終了し、アラームを開始する
・□□□**KT01-05** アラームは、ピッピッピと1秒当たり20msの間、アラームデバイスをONにする

▶ 要求(KT01)に理由が付き、仕様(KT01-xx)が仕様グループごとにぶら下がる ―― この「入れ子」がUSDMの本体

「人間が読める」×「AIが処理しやすい」の2軸で評価

派生開発推進協議会
AFFORDD



★右上の象限に入るのが、AI時代に「使える」記述法

仕様記述 × AI ベストコンボランキング

派生開発推進協議会
AFFORDD

USDM + Gherkin / EARS

汎用最強コンボ ― 要求を構造化＋受入条件をテスト直結

人間が「なぜ」を理解しつつ、AIが受入条件を機械的に展開。コード／テスト生成の精度が安定

USDM + デシジョンテーブル / 状態遷移表

組込み制御系の鉄板 ― ロジック仕様で抜け漏れゼロ

表形式は LLM が分岐を正確に再現できる入力。組込み・制御系で最強の組合せ

OpenAPI / Protobuf + Gherkin

API系プロダクト向け ― スキーマ駆動の一気通貫

スキーマ→実装→テストまで一気通貫。LLMの学習データが豊富で精度プロダクション級

▶ 上位の中心に USDM がいる

⚠ 単独で使うと事故る記述法 — USDM が必要な理由

派生開発推進協議会
AFFORDD

形式仕様単独 (Z, B-method, TLA+)

- ❌ 人間が読めない（人によるが…）
- ❌ 学習コスト高
- ❌ レビュー文化が育たない

数学的厳密性は高いが、組織への定着が困難。運用で破綻する。

ユーザーストーリー単独 (Agile)

- ❌ 情報密度が薄い
- ❌ AI が勝手に補完
- ❌ ハルシネーション温床

「～したい」の一文では、AIは勝手にスコープを広げる。派生開発で致命的。

画像のみの UML / SysML

- ❌ AI が画像を取りこぼす
- ❌ 図と仕様の乖離
- ❌ メンテ負荷が高い

PlantUML / Mermaid 化すれば AI 入力可能。だが組織で徹底できるか？

USDM だけが、人間 ⇔ AI の橋渡しを「構造」として持つ

理由・要求・仕様・グループ — その階層構造が、AI時代の派生開発の共通言語になる

派生開発推進協議会
AFFORDD

AI時代の
新規開発　派生開発

新規開発　AI時代のソフトウェア開発スタイル

派生開発推進協議会
AFFORDD

バイブコーディング（Vibe Coding）

2025年2月、Andrej Karpathy氏（OpenAI共同創設者）が提唱。「vibe＝ノリ」で開発するスタイル。

項目	内容
コンセプト	「雰囲気・ノリ」でAIと対話しながら開発
特徴	コードを書かずに自然言語でアプリを作る
向いている場面	プロトタイプ、小規模開発、UI改善
ツール	Cursor, Windsurf, Claude Code, Replit Agent 等
社会的注目	2025年 Collins辞書「Word of the Year」に選出

課題：AI生成コードの約半数にバグや脆弱性（CSET 2024年11月レポート）
→ スピード重視のあまり、品質・セキュリティが担保されない

仕様駆動開発（Spec-Driven Development）

2025年後半～ GitHub、AWSが提唱。バイブコーディングの問題を解決するアプローチ。

項目	内容
コンセプト	まず「仕様」を明確に定義し、それを基準にコード生成
特徴	仕様が「信頼できる唯一の情報源（SSOT）」
向いている場面	小～中規模開発、複数人/AI協働プロジェクト
ツール	GitHub Spec Kit、AWS Kiro

利点：再現性と透明性が高い／進行がブレにくい／テスト生成と検証の自動化が可能

そうはいっても、ちゃんと仕様を決めないと

派生開発推進協議会
AFFORDD

- AIとのやり取りが続くと、つらい。。。　全体がつかみにくくなる
- トークンばかり消費。。。
- チーム開発では、要求の範囲と担当の範囲が不明確。。。

AIと対話し、USDMで要求仕様を固める

↓

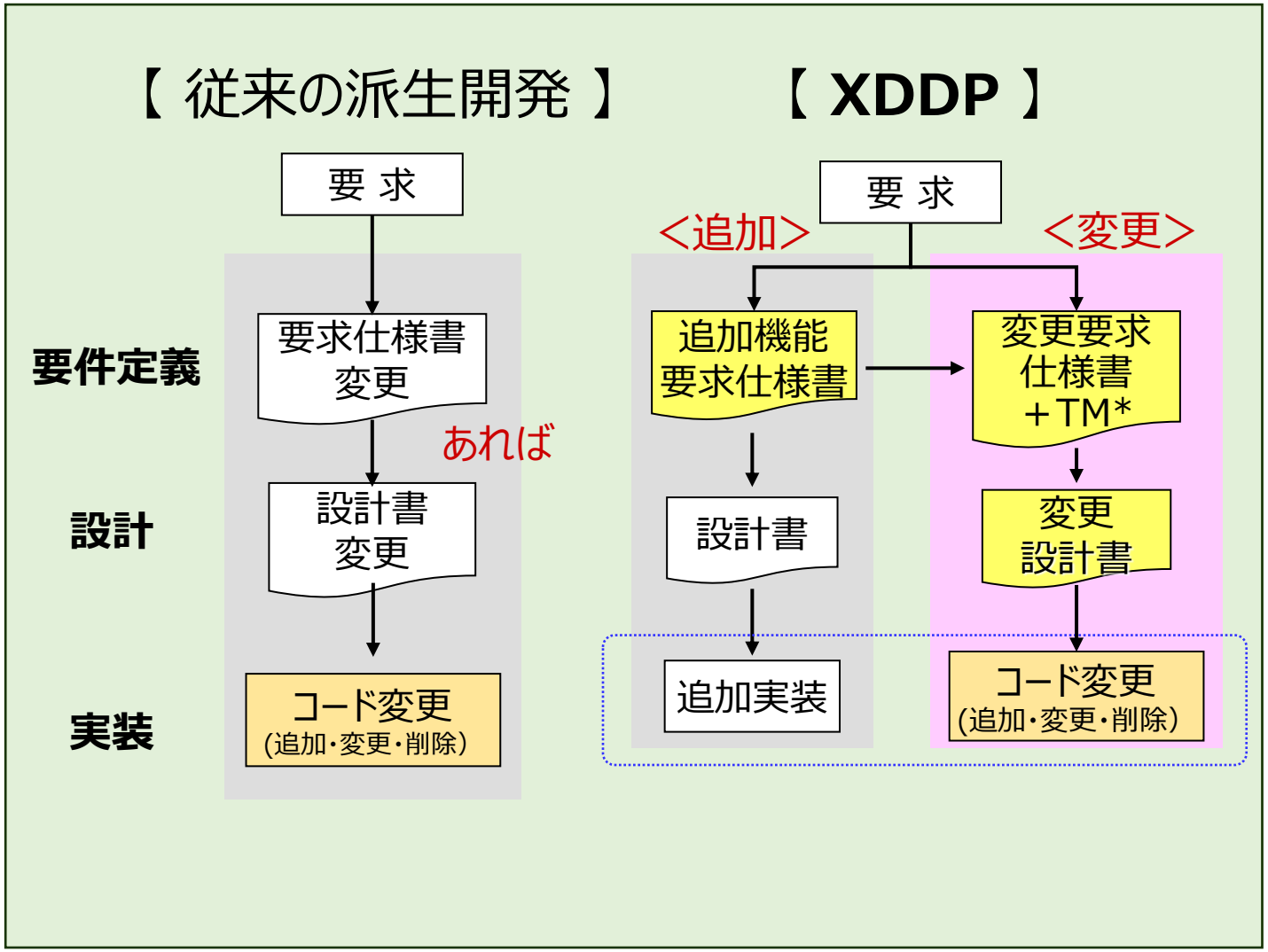
USDMをインプットに、設計、実装を
AIエージェントに任せる

仕様駆動開発はUSDMで駆動しよう！

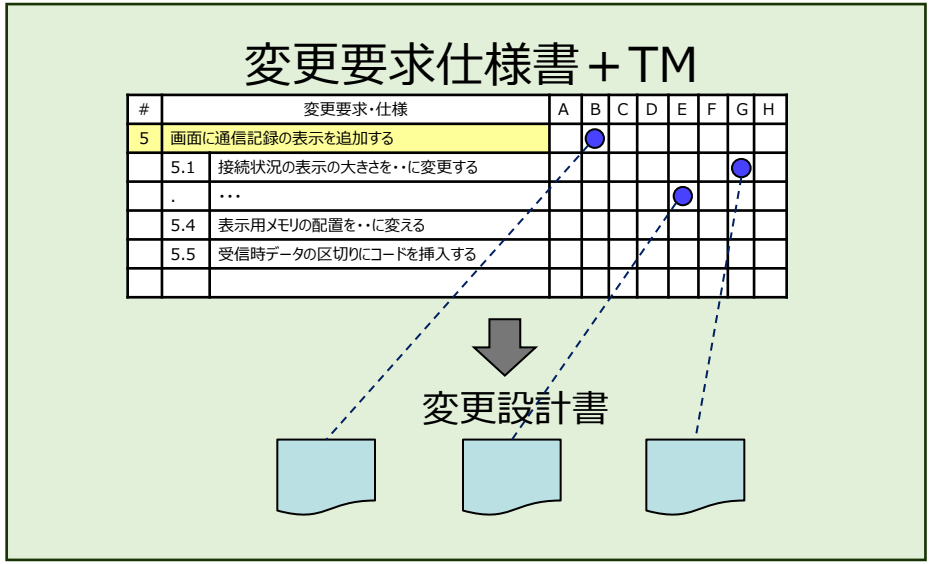
派生開発プロセス　XDDP

派生開発推進協議会
AFFORDD

TM*　：トレーサビリティ・マトリックス

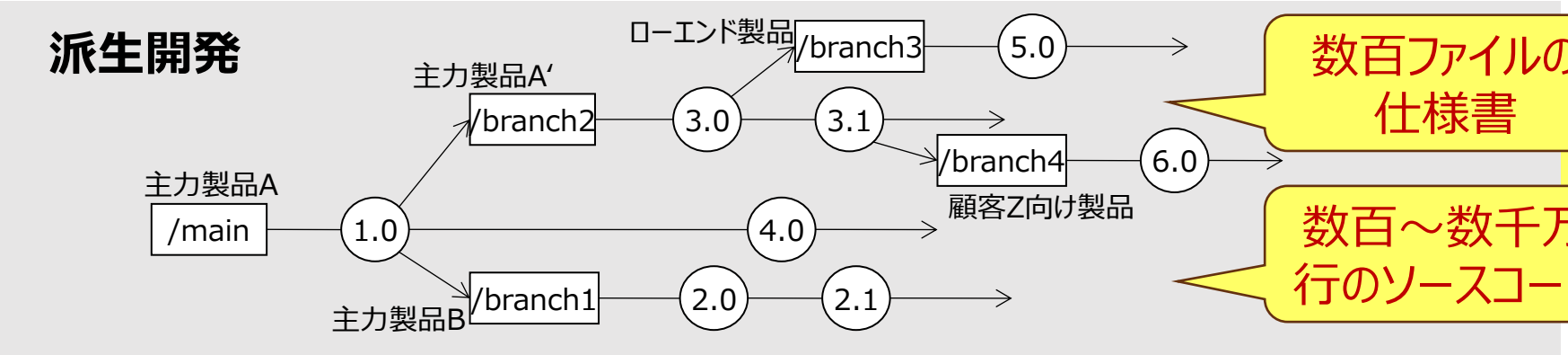


成果物	カバー範囲	内容
変更要求仕様書 (USDM)	*Why What*	何をなぜ変更するか、どのような振る舞いをするか
TM	*Where*	変更仕様がどこにあるか、影響範囲は？
変更設計書	*How*	変更仕様をどのように修正するか



派生開発で流行りのAI活用は適用できる？

派生開発推進協議会
AFFORDD



バイブコーディング（Vibe Coding）

使い物になりません。むしろ、大事故を引き起こす危険なアプローチ

仕様駆動開発（Spec-Driven Development）

既存の仕様が分かっていないと、変更要求や追加要求を駆動できない

- **暗黙知の存在**：仕様書に書かれていない**暗黙知**（歴史的経緯）が**山のように存在**する。**AIはこれを知らない**ため、一見綺麗な、しかし実機では動かないコードを生成する
- **コンテキストの限界**：数百万行の**コード**と関連する**仕様書をすべて**LLMのコンテキストウィンドウに突っ込むことは、現状の技術（数百万トークン対応モデルであっても）では**精度・コスト・速度の面で非現実的**
- **ハルシネーションの致命性**：Webアプリの表示崩れなら笑って直せますが、機器制御のソフトウェアにおけるAIのハルシネーションは、**人命に関わる事故や莫大な損害に直結**する

なぜ AI は派生開発で難しいのか

派生開発推進協議会
AFFORDD

■ 新規開発と派生開発の決定的な違い

- 新規開発：要件をAIに渡せば、白紙からコードが書ける
- 派生開発:「変更の文脈」をAIに渡せていない → 暴走の温床

■ 派生開発で AI に必要な情報は3つ

① なぜ変えるのか（理由）
② 何が、どう変わるのか（Before / After）
③ 何を変え、何を変えないのか（変更すべき箇所）

■ 既存仕様記法の限界

- ユーザーストーリー：情報密度が薄い → AI が補完で暴走
- 形式仕様（Z, B-method等）：人間も AI も扱いづらい
- 自然言語仕様：曖昧で AI が誤解する

■ 変更要求仕様USDM だけが、①～③の3つの情報を構造として持つ

USDM（変更時）の主要マナー　― AI時代を拓く構造

派生開発推進協議会
AFFORDD

①　「理由」欄

＝ LLM のハルシネーション抑制器

理由がないと AI は仕様を勝手に拡大解釈する。

理由があれば「この変更のスコープはここまで」を AI が学習できる。

▶ AIが暴走しない仕様は、必ず『なぜ』を持つ

②　Before/After

＝ 1文に変更を込める記述マナー

❌ before:300円, after:400円（別書き）

⭕ 一律で算出していたのを、時間帯別（具体的な仕様）に変更する（1文）

▶ 差分情報をそのまま AI に渡せる構造

③「変える、変えない」明記

＝ 派生開発の文化を仕様化

「変えないこと」を仕様として書ける記法は、USDM以外にない。

派生開発の暗黙知「触っちゃいけないところ」が構造として可視化される。

▶ 変更要求仕様書とは、変えること、変えないことの合意書である

AI時代の派生開発

派生開発推進協議会
AFFORDD

	小規模開発	大規模開発
Step 1 **変更要求の伝達**	エンジニア → AI 「○○を△△に変更して」	エンジニア → AI 「 ○○を△△に変更して」、 「□□機能を追加して」
Step 2 **AIによる生成**	変更後コード + 変更要求仕様書 + TM + 変更設計書 **一気に生成**	**段階的に生成** 変更要求仕様書 → TM → 変更設計書 → コード生成
Step 3 **エンジニアによるレビュー**	□ 意図が正しいか（変更要求仕様書） □ 影響範囲は適切か（TM） □ 実装方針は妥当か（変更設計書）	□ 意図、変更仕様が正しいか □ 影響範囲は適切か □ 実装方針は妥当か □ コードは正しいか
Step 4 **承認または修正指示**	OK → マージ NG → 修正指示をAIに伝達	OK → マージ

大規模開発で一括生成がダメな理由

派生開発推進協議会
AFFORDD

① レビューの形骸化（LGTMシンドローム）が起きる

- 人間にとって、「ゼロから書くこと」よりも「**AIがもっともらしく書いた複雑なドキュメントの『抜け漏れ』を探すこと**」の方が**圧倒的に困難で苦痛**
- 数百万行のシステムに対する複雑な変更要求仕様、TM（トレーサビリティマトリクス）と変更設計書、そしてコードを**一度に渡された人間は、認知負荷の限界を超える**
- 結果として「**AIがちゃんと追跡しているみたいだからヨシ（LGTM: Looks Good To Me）**」というハンコ押し作業になり、XDDP最大の目的である**「変更時の考慮漏れを防ぐ」という防波堤が決壊**する

② 過程での気づき（暗黙知の抽出）が喪失する

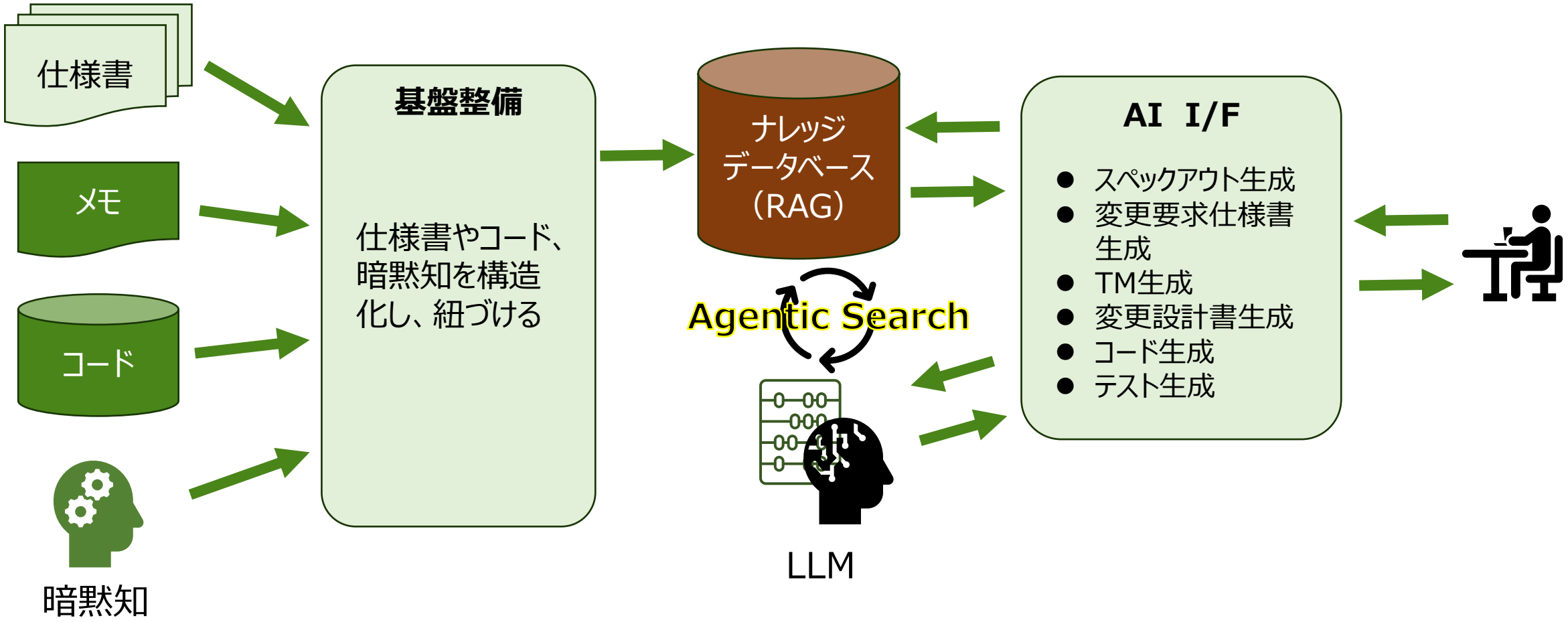
- **XDDPを導入する最大の価値**は、3点セットという「成果物」そのもの以上に、**エンジニアが変更仕様をチェックしたり、変更箇所を特定したりする「プロセス（過程）における気づき」**にある
- 「あ、ここを変更するなら、あの割り込み処理のタイミングも変えないとまずいぞ」という、人間の頭の中にある**暗黙知やドメイン知識は、AIに丸投げした時点で引き出されなくなる**

③ ハルシネーションの連鎖と増幅

- AIが**最初**の「調査」や「変更要求仕様書」の段階でシステムへの理解を**1つでも間違えると**、その間違った前提のままTMを作り、変更設計書を書き、コードを生成する
- 上流工程での**AIの小さな勘違いが、下流に行くほど雪だるま式に巨大なバグとなってコードに埋め込まれる**

AI時代の大規模派生開発戦略

派生開発推進協議会
AFFORDD



本当に必要なのは**コードを速く書けるAI**ではなく**変更すべき場所を正確に教えてくれるAI**

例：駐車場システム「一定料金→時間帯別料金への変更」

Step 1：要件定義担当者が雑な USDM ドラフトを書く

- 「料金体系を時間帯別に変更したい。昼400円、夜200円で。」

Step 2：AI活用で USDM を校閲

- ❌ 要求欄に具体値（400円/200円）が混入　→ USDMマナー違反
- ❌ Before情報（現状300円）が抜けている　→ 変更前の振る舞いが不明
- ❌ 「夜」の時間帯境界が未定義　→ 仕様欄に明記すべき前提が欠落
- ❌ 1日上限の扱いが不明　→ ＜○○の維持＞グループの追加し、仕様欄に「1日の上限金額は変更なし」を明記

Step 3：要件定義担当者が USDM 修正をAIに指示　→　人間レビュー完了

Step 4：AI が派生物を自動生成

- Gherkin Scenario / EARS / TestCase を1:1で導出
- 境界値（07:59 / 08:00 / 19:59 / 20:00）自動列挙
- ＜上限金額の維持＞「変更なし」のテストもAIが自動追加

Step 5：AI が三者間整合をセルフチェック　→　全 Green

これから要件定義担当者に求められる新しいスキル

派生開発推進協議会
AFFORDD

■ USDM が書けるか（ちゃんと確認できるか）どうかが、上流品質を決める

- 派生開発の AI 活用は、要件定義担当者のスキルに依存する

USDM が書けない（確認できない）と

- AIに渡す前の段階で文脈が欠落
  → AIが補完で暴走
- Gherkin/EARS生成も雑、
  テスト導出も不可能
- 人間QAが大量の手戻りを背負う構図

USDM が書ける（確認できる）と

- AIの活用で USDM を自己校閲
  → 完璧な仕様を即座に生成
- Gherkin/EARS は AI が自動導出
  → USDMのモレをフィードバック
- **人間レビューは USDM のみで完結する**

■ USDM は古い記法ではない

- **AI時代の派生開発を生き残るための、新しい必須スキルである**

要件側と検証側を、AIが編み込む

派生開発推進協議会
AFFORDD

プロンプト エンジニアリング

点 ： 1つの指示

例 「タイマー機能を追加。既存の停止処理は壊さない」＝ 要求＋合格条件を1枚に

要件 **意図（Why）**

AI **1つの指示に同居**

検証 **期待結果**

コンテキスト エンジニアリング

面 ： 情報の束

例 既存仕様書＋過去のバグ票を AIの手元にひとまとめに揃える

要件 **仕様・変更履歴**

AI **共有ナレッジRAG**

検証 **不具合・リスク**

ハーネス エンジニアリング

系 ： 流れる仕組み

例 AIが勝手に進まないよう、途中に「設計（ゲート）」を置く。要件と食い違ってたら止める

要件 **トレース自動化**

AI **ゲート統合**

検証 **自動テスト**

ループ エンジニアリング

循環 ： 回り続ける

例 テスト失敗 → 要求へ差し戻し → 再生成 が自動で回り続ける

要件 **逸脱検知**

AI **自律サイクル**

検証 **失敗検知**

検証の学びが プロンプトエンジニアリング へ還る ― 回すほど協働が複利で効く

予防＝要件側 × **検出**＝検証側 → AI ＝ **両者を結ぶ橋**

AFFORDDパートⅠ　クロージング

正しい要求仕様で、AIが正しく応える

USDM は古い技法ではない。
AI時代のソフトウェア開発をUSDMの活用で差を付けよう！