

アーキテクチャ・ハーネス

SimpleModeling

ソフトウェアのアーキテクチャは、求める機能と品質を実現するために、システムを構成する仕組み、その役割、組合せ方を定める基本設計です。SimpleModeling は、ドメイン知識に近い抽象度でソフトウェアを記述し、そのモデルを開発と実行の基盤にするリファレンス・アーキテクチャを提供します。本稿では、連載全 10 回で扱う技術をこの全体像に位置付け、必要な品質を確保しながら開発速度の向上と AI コストの低減を図る組合せを説明します。開発から実行まで何重もの歯止めを配置するこの設計が、AI 開発のアーキテクチャ・ハーネスとしても機能します。

1 アーキテクチャで開発全体を設計する

アーキテクチャでは、求める機能と品質を実現するために、採用する技術とその役割、相互の関係を定めます。SimpleModeling では、その共通設計としてリファレンス・アーキテクチャを用意しています。

モデリング手法だけを取り入れたり、Cozy や Textus を単独で利用したりすることもできます。モデルの検査、Cozy の生成規則、Scala の型検査、Textus の実行規則などは、それぞれが AI の生成・変更・実行を制約するハーネスとして働きます。各要素技術がどのように歯止めをかけるかは、該当する回で説明します。

リファレンス・アーキテクチャは、これらの要素技術を組み合わせた全体最適の設計を示します。例えば、モデルと生成規則で個別実装を減らし、コンポーネントで変更範囲を限定し、型検査でレビューへ持ち込む問題を減らします。さらに、共通の実行基盤で実行時の制約を適用します。この役割分担によって、必要な品質を確保し、開発速度の向上と AI コストの低減を図りながら、AI が要求や規則から逸脱することを抑え、正しい解へ到達しやすい環境を整えます。この組合せがもたらす効果が、本稿で扱うアーキテクチャ・ハーネスの主題です。

1.1 生成・レビュー・修正にかかる時間とコスト

AI を用いた開発では、コード生成に加えて、仕様との照合、レビュー、修正にも時間とコストがかかります。AI によるレビューでは、仕様や周辺コード、依存関係を確認するための推論が必要になり、修正後の再レビューにも推論時間と利用コストが発生します。

生成規則や型による検査は、機械的に判定できる問題をレビュー前に検出します。コンポーネントの境界は、変更と確認の範囲を限定します。これらを組み合わせることで、要求との適合を確認しながら、レビュー対象と修正の反復を減らし、開発時間と AI の利用コストを抑えられます。

1.2 リファレンス・アーキテクチャが歯止めを配置する

SimpleModeling の出発点はモデル駆動開発です。モデルの意味を保ってソフトウェアを構築し実行するために、モデル言語、コンポーネント、プログラミング言語、生成・検査・実行の仕組みを組み合わせます。この共通設計がリファレンス・アーキテクチャです。

AI 開発では、この設計が AI の判断を必要とする範囲と、生成規則や型や実行規則に従う範囲を定めます。ハーネスをアーキテクチャの文脈で捉えたと、作業手順を伝えるスキルもその一部となり、AI が適切な成果を生みやすい開発・実行環境全体が対象になります。

2 SimpleModeling のリファレンス・アーキテクチャの全体像

SimpleModeling のリファレンス・アーキテクチャでは、ドメインの概念・関係・規則を表すモデルを、生成、補完実装、実行、検証の共通の基準にします。CML に記述したモデルから Cozy が構造を導き、AI はその契約の下で不足する処理を補完し、Textus が実行可能なモデルを動かします。CBD でモデルと実装の担当を部品ごとに定め、知識と実行の証拠も同じモデルへ対応付けます。この構成によって、各段階で規則を別々に解釈し直す範囲を減らし、変更や不整合があったときに確認する対象を絞ります。

アーキテクチャ・ハーネス

SimpleModeling のリファレンス・アーキテクチャ

モデルを共通の基準にする

CML：概念・関係・規則
要求と実現モデルを照合

→

生成と補完を組み合わせる

Cozy：構造を生成
CBD+Scala：補完範囲と型

→

同じ規則で実行する

Textus：モデルを直接実行
認可・状態・外部作用を制約

知識・観測・検証を同じモデルと契約へ対応付ける

BoK・RDF・MCPで前提を揃える / 実行の証拠を要求の実例と照合する

単独の検査に残る条件を、別のハーネスで確認する

型が合う → 業務規則も確認 操作が合う → 権限・現在の状態も確認

個別実装・確認範囲・AIレビューの反復を減らす

必要な品質を確保し、開発速度の向上とAIコストの低減を図る

2.1 モデル・生成・補完の共通基準で再実装を減らす

CML はモデルの記述、Cozy はモデルからの構造の導出、Textus は実行を担当します。モデルに書いた規則を Textus が直接実行し、その周辺の接続構造を Cozy が生成することで、規則と接続の両方を AI が個別に書き直す作業を減らします。補完コードにもモデルから導いた型や操作契約を適用すれば、AI がモデルとは別の前提で実装する余地を狭められます。

CBD は公開された操作と利用条件を持つ部品を定め、3 層の構成は各部品の役割を整理します。画面や対話はプレゼンテーション、ユースケースに沿う処理の組合せはアプリケーション、対象の概念・状態・規則はドメインが担当します。ドメインは静的モデル、アプリケーションはユースケースを起点とするワークフローを中心とします。この二つの区分を組み合わせると、画面を増やす場合も既存の業務操作を利用でき、規則を画面ごとに再実装・再検証する負担を減らせます。

2.2 実行経路とモデルの共有で重複と調査を減らす

アプリケーションのワークフローがドメインの公開操作を呼び出し、状態機械や業務規則に従って処理を進めます。CLI、REST、MCP はその同じ処理を呼び出す入口にします。業務ロジックと認可・外部資源へのアクセスを共通の経路へ集約することで、入口ごとに異なる判定が生じる余地を減らせます。共通部分の仕様と検証結果を再利用し、入口固有の入力変換や認証情報の引渡しは個別に確認します。

開発に使う知識は用語・モデル・仕様との対応と出典・版を持たせ、実行結果は同じモデルの語彙へ戻します。この対応を要求の検証と組み合わせることで、結果が合わないときに、使った知識、記述した規則、実行した処理のどこを確認すべきかを調べられます。モデル、ログ、文書を別々に読み直して関係を推測する負担を減らし、根拠のある修正へつなげます。

2.3 全 10 回の技術を全体の役割へ位置付ける

第 1 回は、AI 開発の可能性と業務アプリケーションに必要な条件を整理し、検証・修正・承認までを含む収束コストを扱います。第 5 回は、その条件を満たすために各回の技術を組み合わせる設計を扱います。

第 2 回のモデル・ハーネスは、CML、モデル駆動開発、モデル・アップ・ダウン、要求と実現の対応を基盤にします。第 3 回はエンティティ、値オブジェクト、集約、関連、パワータイプ、トレイト、リソース、タスクの分類や制約などの静的モデル、第 4 回はワークフロー、状態機械、ユースケースなどの動的モデルを扱います。対象の操作や状態と手順を突き合わせて不整合を検出し、実装前の修正につなげます。

第 6 回は Scala の型、代数的データ型、イミュータブルなデータ、純粋関数型プログラミング、高階関数、モナディックな計算構造、内部 DSL、Free Monad とインタープリタで補完実装を制約します。第 7 回は Textus によるモデルの実行と、イベント、認可、外部作用、並行性、再試行・補償を扱います。言語側で許す操作を実行基盤の経路へ接続し、実行前に判定できる問題は型などで検出し、利用者や実行時の状態に依存する条件はランタイムで検査します。両者を組み合わせることで、AI の補完コードごとに権限や失敗制御を作り込む範囲を減らします。

第 8 回の知識ハーネスは、BoK の用語集を鍵に文芸モデルとオブジェクト・モデルを関連付け、RDF で出典・版・適用範囲を管理し、MCP から AI へ提供する仕組みを扱います。対象のモデルや部品の契約と知識を揃えることで、AI が古い仕様や別の利用条件に基づいて実装し、後から大きく修正する事態を減らすことを目指します。

第 9 回の観測モデルは実行時エビデンスをモデルの語彙と抽象度へ戻し、第 10 回の検証ハーネスは動く仕様書、テスト、AI レビュー、人間の判断・承認を扱います。期待する結果と実際の操作・遷移を対応付ければ、結果が違った事実に加えて、どこを修正するかを判断できます。各要素技術の検

査と診断を利用して、原因に応じた修正を行います。個別の仕組みは担当回へ委ね、本稿はその相互作用を説明します。

3 高い抽象度のモデルを実行の基準にする

モデルには、ドメインの概念・関係・規則を、業務知識に近い抽象度で記述します。その構造を検査と実行にも利用することで、同じ規則を個々のコードへ翻訳する作業と、モデルとの照合を減らせます。

3.1 DDD レベルのモデルと直接実行を組み合わせる

ドメイン駆動設計（DDD）で扱うエンティティや値、集約、操作を、業務の意味が分かる構成要素としてモデル化します。動的な処理はワークフローの順序・分岐、状態機械の状態・イベント・遷移条件として記述します。Textus が実行可能なモデルを直接動かすことで、実行の進行もこの定義に従います。

規則を個別のフラグや条件分岐へ書き直す範囲が減るため、AI による補完と、モデルとコードの照合作業を減らせます。高い抽象度で記述したモデルが検査・実行の基準として働くことが、モデルを AI 時代のプログラムとして扱える理由です。

3.2 要求の検査と規則の評価を実行モデルへ結び付ける

ユースケースは、利用者の目的と実例を表す要求モデルです。ユースケース自体を直接実行するのではなく、それを実現するワークフロー、状態機械、ドメインの操作へ対応付けます。textus-cbd-support によるモデルのレビューで実現対応と構造の整合を確認し、動く仕様書で同じ実例の実行結果を確かめます。この二つを組み合わせることで、要求に対する実現の不足をモデル段階で探し、実行時にその意図どおりの結果が得られるかを確認できます。

業務規則はドメイン側のモデルとしてまとめ、ルール・エンジンに評価を委ねる構成を取れます。規則の所有場所を CBD で定め、評価方法を共通化すれば、画面や操作ごとに別の判定処理を書く余地を減らせます。CNCF にはルール・エンジンの基盤がありますが、ドメイン・ルールをモデルの第一級要素として実行系へ対応付ける部分は設計課題として残ります。

3.3 生成と AI による補完の担当を分ける

モデルから機械的に導けるインターフェースや接続構造は Cozy が生成します。同じ規則で構造を導くことで、AI が機能ごとに異なる構成を選び、それを後からレビューして揃える作業を減らします。

モデルだけでは表現しきれない固有の計算や外部サービスとの接続は、AI がプログラミング言語で補完します。開発者はドメインのモデルを中心に扱い、補完部分には生成された型や操作契約とコンポーネントの利用規則を適用します。モデルが変わった場合も、構造を再生成し、型や契約と合わなくなった補完箇所を検査して修正できます。生成と検査を組み合わせることで、AI が変更箇所を推測して全体を書き換える範囲を減らします。

4 CBD と Scala で補完実装とレビューを局所化する

モデルを直接実行する構成では、モデルに記述できない処理をコードで補完し、そのコードと部品間の接続を検査します。CBD で担当範囲を区切り、その内部に Scala の型や計算の規律を適用することで、部品間の不整合と局所的な実装の誤りをそれぞれ確認できます。

4.1 公開契約が変更と確認の範囲を限定する

各コンポーネントには、提供する能力を表すカーパビリティと、公開するサービス・操作、その入出力や利用条件を定めます。外部は公開 API を使い、差し替え可能な実装は SPI に従います。内部データへ直接依存させず、許された依存関係を定めることで、内部実装の変更が周囲へ波及する経路を減らします。

公開契約をモデルから導く型や動く仕様書と結び付けると、部品内部を AI が変更しても、利用側との接続と期待する振る舞いを別々に確認できます。契約を保つ変更では担当部品とその契約の確認を中心にでき、システム全体を毎回読み直す負担を抑えられます。契約自体を変える場合は依存関係から利用側を調べ、必要な確認範囲を広げます。

4.2 Scala の検査能力をレビューの前に働かせる

Scala の型検査は、それ自体が補完実装に対するハーネスです。これをモデルから導く型や CBD の公開契約と組み合わせると、AI が別の意味の値を渡したり、契約に合わない結果を返したりする問題のうち、型に表せるものをレビュー前に検出できます。代数的データ型で操作の結果や状態を表し、必要な場合分けの不足もコンパイラの診断へ結び付けます。

イミュータブルなデータと純粋関数は、値の書換えや副作用の発生箇所を限定します。CBD で狭めた部品内部を、入力と出力、明示的な作用に分けて理解できるため、レビューの際に追跡する状態と実行経路を減らせます。

この組合せによって、機械的に検査できる不具合を理由とするレビューと修正の反復を抑え、開発速度の向上と AI コストの低減を図ります。これは設計上の効果であり、削減率の実測を示すものではありません。要求や業務上の意味が正しいかは、別途モデルと実例に基づいて確認します。

4.3 操作の語彙を実行基盤の制約へつなぐ

内部 DSL は、プログラムが利用する操作の語彙を定めます。Free Monad では、操作の組合せを値として記述し、インタープリタが外部への作用を実行します。モデルで定めた操作、Scala の型、DSL、共通の実行機能を対応付けることで、任意の場所から外部資源へアクセスする実装を減らせます。

DSL で許された操作が認可や資源アクセスの共通経路を通るようにすると、操作を記述するコードと、その作用を許可する仕組みを分担できます。権限や制限を補完コードの各所へ書く必要が減り、方針変更時の修正箇所もまとめられます。型や DSL による制約は第 6 回、実行時の条件と迂回経路の検査は第 7 回で扱います。

5 複数のハーネスで制約の抜けを補う

各ハーネスが制約できる範囲には限りがあります。例えば、型検査を通る実装でも業務規則には違反する場合があります、モデルに整合する操作でも実行時の権限や状態によっては許可できません。モデル検査、型検査、契約の検査、実行時の制約、要求に基づく検証を重ねることで、一つのハーネスでは扱えない条件を別のハーネスで確認し、制約の抜けを補います。

リファレンス・アーキテクチャでは、各ハーネスの適用範囲と、残る条件を確認する機構との対応を定めます。モデル検査で構造や対応、型検査で値や操作の整合を確認し、実行時には利用者の権限や現在の状態を確認します。実行結果は要求の実例と照合します。機械的に確認できる問題を早期に処理し、後段で残る条件を確認する配置によって、多重の制約と開発効率を両立させます。

検査対象のモデル、契約、実装には、対応する版と適用条件を指定します。対象が変わった場合は、依存関係から影響する検査を特定して再実行します。前段の結果を利用できる範囲を明確にすることで、重複した確認を減らし、変更後に必要な検査を実施できます。

5.1 部品と組合せの検査で修正対象を絞る

モデルの構造や対応を確認してから、そのモデルに基づく生成と型検査へ進みます。モデル側に原因がある診断はモデルへ戻し、補完コードの型違反は担当部品で修正します。この連携によって、誤ったモデルから生成した多数の箇所を個別に直す作業や、型で判定できる問題を AI が繰り返しレビューする作業を減らせます。

CAR は実装・契約・依存・設定・仕様を同じ配布・検査単位へまとめ、SAR は複数部品の構成を表します。これにパッケージ検証と lint を適用し、必要な構造や規則化できる依存・禁止パターンを調べます。単体の部品の成立と、組み合わせた際の依存や設定の整合を分けて確認し、その結果を受理条件に使います。検査で不整合が見つかった部品や組合せを修正対象として示せるため、後から構成全体を調べ直す負担を抑えます。

5.2 実行時にも権限と状態の規則を適用する

静的検査は操作の接続や型を確認し、実行時の検査は利用者の権限や現在の状態を確認します。公開操作の実行経路に認可とワークフロー・状態機械の判定を組み込み、権限や遷移条件に反する呼出しを拒否します。構築時と実行時の判定を分担することで、AI がこれらの条件を呼出し元ごとに実装する範囲を減らせます。

外部資源への作用には、CAR lint による早期検出、読み込み時の受理検査、実行コンテキストに応じたサンドボックスを組み合わせる設計を置きます。CBD の区分に資源アクセスの制約を対応させることで、誤変更や誤操作の影響を限定します。悪意あるコードの隔離には、別途プロセスやコンテナの隔離が必要です。具体的な適用範囲と実装状況は第 7 回で扱います。

5.3 知識と契約を揃えて判断の前提と範囲を限定する

実装の前提となる業務知識についても、対象の版と適用条件を確認します。知識ハーネスでは BoK の用語集を介して文芸モデルとオブジェクト・モデルを関連付け、RDF で出典・版・適用範囲を保持し、MCP で提供する知識を開発対象に揃えることを目指します。これを CBD の契約と組み合わせれば、AI に担当部品に関係する知識と利用条件を渡し、仕様を探索・再解釈する範囲と、前提の誤りによる手戻りを抑えられます。

開発用のスキルには、参照するモデルと契約、変更できる範囲、検査結果に応じた対応を示します。知識の整合で判断の前提を揃え、生成・検査・実行の制約で成果を確認します。スキルが示す手順と処理系の検査を対応付けることで、手順からの逸脱を検出できます。アプリケーション内で AI を利用する場合も、曖昧な情報の解釈や候補生成を AI が担当し、定型的な進行と状態変更はワークフローや実行基盤が担当します。これにより、処理のたびに AI が手順全体を組み立て直す推論と、その逸脱の機会を減らします。

5.4 要求と観測を対応付けて修正先を判断する

観測モデルは、実行された操作・状態遷移・失敗などのエビデンスをドメインやユースケースの語彙へ対応付けます。要求と実現の対応にこの観測を重ねることで、期待した振る舞いのどこで実際の進行が異なったかを確認できます。モデルアップで判断できる抽象度へ戻すため、ログとコードの関係を毎回読み解く負担を減らし、モデルや担当部品の修正へつなげられます。

動く仕様書は、入出力、前提、実例、失敗時の期待を実行して確かめる契約です。CLI、REST、MCP で同じ操作の仕様を共有し、AI による意図の解釈は別の適合性テストで調べます。失敗箇所を切り分けた上で、意味の妥当性や例外の受容を AI レビューと人間の判断で確認し、承認した変更をモデルや知識へ反映します。

6 注文処理で組合せの働きを確認する

在庫確保と支払い承認を経て注文を確定する処理を、組合せを説明する設計例として用います。要求、モデル、補完実装、実行、検証の対応を順に示します。

6.1 要求とモデルから部品の操作を定める

ユースケースに購入者の目的と、在庫不足・支払い失敗を含む実例を記述します。注文・明細・金額を静的モデル、在庫確保・支払い承認・注文確定をワークフロー、注文の状態と遷移条件を状態機械として定義します。要求した結果へ至る操作が揃うかをモデル上で確認します。

価格や税額の規則はドメイン側が所有し、アプリケーションのワークフローは価格を求める公開操作を利用します。規則をルール・エンジンへ接続する構成なら、その評価も当該操作の内側へ集約します。規則モデルと所有部品と利用操作を揃えることが、画面ごとの別計算を防ぐ設計になります。

6.2 補完コードを検査して同じ処理を実行する

Cozy が操作の接続構造を生成し、AI は支払いサービスへの接続などを補完します。金額・注文識別子・処理結果を型で区別し、CAR の検証と lint で依存や操作の利用規則を確認します。例えば補完処理が誤った型の結果を返す問題はコンパイルで検出し、別部品の内部へ依存する問題は規則化した範囲で lint 等により調べます。どちらも同じ補完処理を対象にするため、診断された箇所を修正し、接続と依存の整合を再確認できます。

実行時には、利用者が注文を確定できるかを認可し、Textus がワークフローと状態機械に従って進行させます。支払いが失敗した場合は確定へ進めず、定義した失敗処理や在庫予約の補償へ進みます。外部サービスの呼出しも共通の作用経路に置くことで、型検査だけでは決まらない権限や実行条件を適用できます。

6.3 証拠を使って修正対象を絞る

動く仕様書で、正常な注文確定と支払い失敗時の結果を確かめます。観測モデルには、どの操作と遷移を経たかを示します。実行順序の問題ならワークフロー、金額計算なら規則や補完コード、利用者権限ならポリシーというように、証拠に基づいて修正対象を絞ります。

この注文処理では、金額の型が正しくても、適用した価格や税額の規則に誤りがある場合があります。規則のモデルと要求の実例を照合し、動く仕様書で計算結果を確認します。また、注文確定の呼出しが型や契約に合っている、支払いが未承認であれば状態やワークフローの条件で止め、利用者に権限がなければ認可で止めます。同じ処理へ異なる観点の制約を重ねることで、一つの検査を通過した処理に残る不整合や許可条件の抜けを補います。

7 品質と効率を両立させるアーキテクチャ・ハーネス

SimpleModeling のリファレンス・アーキテクチャは、モデルを中心に、CBD、Scala、生成・検査・実行基盤、知識、観測、検証を組み合わせます。各技術の担当を定めることで、必要な品質を確保しながら、開発速度の向上と AI コストの低減を図ります。何重もの歯止めの配置も、この全体最適の設計に含まれます。

7.1 予防・局所化・共通化によって反復を減らす

個々のハーネスは、モデル、部品、実装、実行などの範囲で誤りを防ぎます。リファレンス・アーキテクチャでは、その間を同じモデルや契約で結びます。直接実行と生成によって減らした補完コードを CBD で区切り、その内側を型で検査し、許された操作を共通基盤で実行します。個別実装、確認範囲、レビューへ持ち込む問題がそれぞれ減るため、各段階で行う AI の推論と修正の反復を抑えられます。

各機構の役割分担に、単独では残る制約の抜けを相互に補う関係を組み合わせます。機械的に確認できる条件を生成・検査・実行基盤で扱い、意味や要求との適合をレビューで確認することで、確認範囲を補いながら AI による推論と修正の反復を抑えます。この構成は、構造の維持、誤変更の影響

の限定、公開仕様に基づく再実装にも役立ちます。効果の確認には、検証・修正・再レビュー・承認までの時間と AI 利用量を用います。

診断と実行の証拠は、AI による修正対象の特定にも利用します。モデルの誤りはモデルへ、型の不整合は補完コードへ、権限の不整合は呼出し条件やポリシーの確認へ戻し、原因に応じた修正を行います。要求や契約自体の変更には、実装の修正とは別の判断・承認を置きます。診断に対応する修正範囲と確認事項を指定することで、AI が無関係な箇所へ修正を広げることを抑えます。

7.2 各回で仕組みを具体化する

各要素技術のハーネスとしての働きは、第 2～4 回のモデル、第 6 回の Scala、第 7 回の実行基盤、第 8 回の知識、第 9 回の観測、第 10 回の検証で説明します。リファレンス・アーキテクチャは、これらの間でモデルや契約を共有し、実装・推論・レビューの重複を減らす構成を定めます。各機構の制約と診断によって、AI の変更範囲を限定し、要求に適合する成果へ修正を進める環境を整えます。

CNCF では、DDD レベルのモデルを実行する構成に、リアクティブな処理、高可用性、セキュリティ、AI の利用を組み合わせ、業務システムの品質要求へ対応させます。その具体的な構成や実装段階は後続の CNCF シリーズで扱います。