

プログラミング・ハーネス

SimpleModeling

SimpleModeling では、Scala を中心としたプログラミングの仕組みを、AI による開発のハーネスとして利用します。第 6 回では、モデルの直接実行とコード生成を前提に、AI が補う連携コードや業務固有の処理を、型・合成規則・共通 API・検査によってどう支えるかを説明します。

1 プログラミング・ハーネスの役割

プログラミング・ハーネスは、AI が使う実装の構造や操作を定め、誤りや逸脱を抑える仕組みです。値の取り違い、共有状態への依存、失敗時の扱い漏れなどを、個別の注意だけに頼らず防ぐことを目的とします。

既存機能を適切に再利用することで、AI が独自に設計・実装する範囲を減らします。追加が必要なコードには、誤りが入りにくい構造と適合を確認する条件を与えます。この二つの働きによって、AI の判断と修正の負担を抑えます。

2 SimpleModeling でのプログラミング・ハーネスの全体像

第 5 回で示したりファレンス・アーキテクチャでは、モデル、生成コード、共通の実行基盤、AI の補完コードが役割を分担します。ここでは、モデルで定めた仕様が型付き API を介して補完コードへ引き継がれる流れを示します。

2.1 モデル由来の型から再利用とグルーコードへつなぐ

CML で記述したモデルの実行可能な部分は Textus が直接実行し、Cozy はモデルから型付き API や実装の接続構造を生成します。型に入出力や接続条件が明示されるため、AI は利用可能な既存機能とその使い方を判断しやすくなります。

AI が補うのは、部品をつなぐグルーコードと、既存機能では不足する業務固有の処理です。型が示す情報を手掛かりに変換や連携を記述でき、不適合な接続はコンパイラが検出します。型による仕様の明確化が、再利用と補完コードの作成を支えます。

2.2 補完コードから共通基盤の操作を利用する

保存、外部通信、非同期実行や障害対応などを支える機構は共通基盤が担い、補完コードは公開された API を通じて利用します。業務処理の記述と実行管理を分担した上で、補完コードに型、純粋関数と合成規則、部品の検査を適用します。以降では、それぞれが担当する制約を具体的に説明します。

3 値の取り違えと扱い漏れを型で検出する

型によって、同じ形式の値を業務上の意味で区別し、処理結果として取り得る種類を定められます。値の取り違えと場合分けの不足を例に、AI が生成するコードへどのような条件を適用できるかを見ます。

3.1 番号や数量の意味をプログラムでも保つ

顧客番号と注文番号をどちらも文字列にすると、引数を取り違えても型検査を通ります。Scala の opaque type などでは CustomerId と OrderId を区別すれば、異なる型の値を渡す誤りをコンパイラが検出できます。

数量の範囲などの条件は、値の生成時に検査し、成功した値だけを専用の型として渡します。その生成経路を維持し、後続の関数が専用の型を要求することで、確認済みの条件を引き継ぎます。AI が書くコードも同じ型を使うため、値の誤用を検出し、個別の重複チェックを減らせます。

3.2 状態や結果の種類を扱い漏らさない

処理結果の種類を代数的データ型 (ADT) で定め、パターンマッチで場合分けすると、新しい結果を追加した際の扱い漏れを検出できます。例えば、在庫予約に「受付停止」を加えたとき、網羅性の警告をビルドエラーとして扱う設定によって、対応していない呼出し箇所を確認できます。

何でも受け取る場合分けは追加を隠すため、レビューの対象にします。AI が成功時の処理だけを生成した場合も、網羅性検査で不足を返せます。状態の種類も型で表せますが、許される状態遷移は状態機械の規則で確認します。

3.3 型による証明を実務の検査へつなぐ

カーリー＝ハワード対応は、型を命題、型を持つプログラムをその命題の証明として捉える考え方です。基本的には、直観主義命題論理の証明と単純型付きラムダ計算のプログラムが対応します。関数の型 $A \rightarrow B$ は「A が成り立つなら B が成り立つ」という命題に対応します。

これらの例では、業務上の区別や接続条件を型へ表し、その条件を満たさないコードをコンパイラで検出しています。型で保証できる範囲が広がるほど、個別の検証や AI レビューで繰り返し確認する条件を減らせます。通常の Scala の型検査で停止性や業務全体の振る舞いまで証明するわけではありませんが、生成・修正のたびに同じ条件を機械的に適用できることは、実務上の効果になります。

参考文献：Philip Wadler, Propositions as Types, Communications of the ACM 58(12), 2015。
<https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf>

4 純粋関数と合成規則で実装の構造を制約する

型で値や接続の条件を確認しても、処理の順序、状態への依存、失敗の扱いは記述する必要があります。計算を純粋関数で表し、処理を共通の合成規則で組み合わせることで、この部分の構造にも制約を与えます。

4.1 高度な合成規則を採用する理由

クラウド・ネイティブなアプリケーションでは、負荷の変動や部分的な障害に対応しながら応答を維持する必要があります。リアクティブ宣言は、そのための四つの特性として、応答性、耐障害性、弾力性、メッセージ駆動を整理しています (<https://www.reactivemanifesto.org/>)。

非同期処理や障害対応を業務コードごとに実装すると、処理の順序や失敗時の制御を個別に管理する必要があります。共通基盤が提供する型付きの操作を、モノドとアプリカティブによって依存関係や独立性に沿って組み合わせることで、複雑な実行制御を個別に記述する範囲を減らします。リアクティブな特性を支えるのは実行基盤であり、合成規則はその機構を業務コードから利用するために用います。

純粋関数型プログラミング、モノド、アプリカティブは習得や記述に難しさがありますが、AI がコードの記述を担うことで開発へ取り入れやすくなります。その出力を型と共通 API、検査で支えることで、高度な手法の利点を実務で利用できます。個別の制御実装と修正・再レビューの負担を抑え、必要な品質を確保するための開発時間と AI 利用コストの低減を図ります。

4.2 計算に隠れた状態への依存を持ち込ませない

関数の内部で現在時刻や変化する価格表を参照すると、同じ引数でも結果が変わります。必要な日時、価格表、規則の版を引数へ集め、データを不変に保つことで、同じ入力から同じ結果を得る純粋関数として計算できます。業務規則の評価でも、事実を取得する処理と、固定した事実規則を適用する計算を分けます。

入力を明示し、不変データを用いる構造に揃えると、AI が計算の途中で共有状態へ依存したり、状態を書き換えたりする実装を避けやすくなります。Scala の `val` や関数の型だけでは純粋性は保証されないため、使用する API や依存関係を検査・レビューで確認します。この構造の指定と適合確認を組み合わせることで、隠れた状態に依存する誤りを抑え、修正も関数と入力へ局所化できます。

4.3 Consequence で結果とエラー処理の規則を揃える

Textus/CNCF では、関数の処理結果を Consequence で表します。正常な結果には値の型を持たせ、失敗にはエラーの分類、原因、対処の方向などを構造化した Conclusion を保持します。結果の接続とエラーの解釈に共通の体系を適用するため、AI が関数ごとに異なる失敗の表現を作ることを抑えられます。

前段の結果に依存する処理は、Consequence のモノドとしての合成を使います。flatMap は成功した値を次の処理へ渡し、失敗した場合は後続処理を進めずに失敗を引き継ぎます。型が処理の接続を確認し、共通の合成規則が失敗の伝播を担うため、AI が分岐を個別に書き直す必要が減ります。

互いの結果に依存しない入力検査には、Consequence の zip 系 API によるアプリカティブな合成を使い、各項目のエラーを集約します。依存する処理では失敗時に止め、独立した検査ではエラーをまとめるという違いを、API の選択で表します。設計・レビューでは、この選択が業務上必要な失敗の扱いに合うかを確認します。アプリカティブな合成は独立性を表すもので、並列実行そのものを保証するものではありません。

5 UoW と Free Monad で操作の記述と実行を分ける

Consequence で結果と失敗の扱いを揃えても、保存や外部呼出しをどこで実行し、その実行を誰が管理するかは別に定める必要があります。Textus/CNCF では、UoW (Unit of Work) が一連の操作の実行単位と共通の実行管理を担います。Free Monad で操作を記述・合成し、実際の実行を UoW の共通機構へ渡すことで、業務処理の組立てと実行管理を分けます。

5.1 UoW の操作を型付きの内部 DSL で記述する

UoW で実行する保存や外部呼出しには、操作ごとに入力と結果の型を定めます。内部 DSL は、これらの型付き操作を Scala の中で記述し、組み合わせるための API です。AI は公開された操作を利用して、部品の連携や業務固有の処理を記述します。

操作の種類と接続条件を API に表すことで、AI が使う実装手段を具体化できます。型検査は不適合な接続を検出します。共通の実行経路を外れる直接呼出しなど、型だけでは判別できない逸脱は、後述する依存関係の検査やレビューで確認します。

5.2 UoW で実行する処理を Free Monad と Consequence で組み立てる

Free Monad では、UoW で実行する操作を、その場で実行せずにプログラムの記述として組み合わせます。操作の入出力と接続条件を型で検査し、この記述を組み立てる関数を純粋に保つことで、組立て中の副作用や隠れた状態への依存を抑えます。この組合せによって、外部操作を含むプログラムにも、型による接続の保証と、純粋関数として局所的に推論できる構造を適用します。

この操作の記述に Consequence の合成規則を組み込むことで、計算と外部操作を同じ結果・失敗の体系でつなぎます。Free Monad は操作の記述と合成、Consequence は結果と失敗の扱いを担当します。実装上は、操作を UnitOfWorkOp、Free Monad によるプログラムを Program で表し、ConsequenceT によって Consequence の結果・エラーの規則をプログラムへ組み込んでいます。

インタープリタは記述されたプログラムを解釈し、UoW の共通実行経路で操作を実行して、結果を Consequence で返します。操作の組合せや実際の応答はテストで確認します。非同期実行、通信、障害対応、資源管理などを含む実行基盤がリアクティブな特性を支え、その詳細は第 7 回で扱います。

型付きの操作を純粋に組み立て、結果と失敗を Consequence で扱い、実行管理を UoW へ委ねます。この分担によって AI が個別に作り込む制御を減らし、検査・レビューで分担への適合を確認することで、プログラミング・ハーネスとして機能します。

6 部品の構造と業務上の意味を検査する

型と操作の規則に適合したコードについても、部品の責務や要求との整合性を確認します。CAR の構造検査、依存関係の確認、テスト、モデルに基づくレビューが、言語の型だけでは扱えない条件を補います。

6.1 CAR の契約で部品構成からの逸脱を検査する

CAR は、Component、Service、Operation などの契約と、実装、依存関係、説明を検査可能な構造にまとめます。car-lint がその構造と公開契約について機械化された規則を確認することで、AI が追加したコードを部品としての適合検査へ載せられます。型の合う関数を作るだけでなく、決めた部品構成へ収める条件がハーネスとして働きます。

textus-cbd-support は、部品の仕様や依存関係、モデルの情報を参照したレビューを支えます。car-lint の構造検査に加え、こうした情報を使って実装が部品の責務に合っているかを確認します。

6.2 共通機構が担う処理を個別実装から外す

共通機構が担う処理を AI が重ねて実装していないかも確認します。検査済みの入力のリ再検査や、基盤が管理する処理への独自の再試行は、不要な分岐や二重実行を生むことがあります。入力の型と標準 API の契約を確認の基準にします。

依存関係の検査では、公開 API を経由しない呼出しを確認します。レビューでは、追加した防御処理が業務上必要か、共通機構の保証と重複していないかを確認します。これにより、必要な業務上の確認を保ちながら、定めた実装構造への適合を検証します。

6.3 機械的な適合と業務上の意味を分担して確認する

型が合う金額計算でも、割引率や端数処理が要求と違えば業務上は誤りです。型検査は値と接続、lint は構成規則、テストは具体的な入力に対する結果、モデルに基づくレビューは要求と設計の対応を確認します。異なる条件を確認する検査を組み合わせ、一つでは捉えられない誤りを補います。

7 診断を修正へつなぎ、変更の発散を抑える

検査で得た診断に基づき、修正する範囲と再確認する条件を定めます。修正先の選択と修正後の確認を組み合わせ、AI による変更の発散を抑えます。

7.1 変更の原因に応じた箇所を直す

修正は原因のある箇所へ戻します。業務上の意味はモデルへ、生成する構造は生成規則へ、固有の処理は補完コードへ反映します。生成結果だけの修正が次の生成で失われることを防ぎ、変更の責任を明確にします。

モデル変更後は型と API を再生成してコンパイルし、接続が合わなくなった箇所を特定します。AI は契約を緩めて診断を消すのではなく、変更された契約へ補完コードを適合させます。

7.2 診断を局所修正へ返し、判断が必要なら戻す

コンパイル、lint、対象を絞ったテストで診断を得て、該当箇所を修正し、同じ条件を再確認します。再現できる診断と確認条件を AI へ渡すことで、修正の対象を限定できます。

原因が特定できない場合や契約の変更が必要な場合は、変更を増やし続けず、モデルの情報を添えて設計の検討や担当者の判断へ戻します。機械的検査、意味のレビュー、人間の判断を分担することが、修正の発散を抑える条件になります。

8 まとめ

型、合成規則、共通基盤、部品の検査を組み合わせることで、AIの実装を誤りが入りにくい構造へ導き、再利用と補完の両方に制約を適用できます。AIの記述能力をこの枠組みで利用することにより、高度な手法の利点を取り入れながら修正や再レビューの負担を抑え、必要な品質を確保しつつ開発時間とAI利用コストの低減を図ります。

次回の第7回では、実行時の認可や資源管理を支えるランタイム・ハーネスを扱います。