

Static Model Harness — Describing Domain Structure

2026-10-05
SimpleModeling

The Static Model Harness provides a foundation for describing domain concepts, relationships, and rules as models and connecting them to checking and execution. Part 3 of this series explains how established software engineering practices can make business constraints effective during AI-assisted generation and modification.

1 The Role of the Static Model Harness

1.1 The Place of This Installment

Part 1 introduced the AI Development Harness as a whole. Part 2 examined the Model Harness built on model-driven development. Part 3 addresses its static aspect: the concepts, types, identity, relationships, ownership, and constraints that constitute software. Part 4 will turn to events, state transitions, and workflows for the same objects.

Static and dynamic models describe the structure and behavior of the same model. This distinction is separate from the distinction in purpose and concerns between domain models and application models. Domains have behavior, and applications also have static structure.

1.2 Building on the Previous Series

The previous series, in “Model Structure and Views” and “Object Modeling as a Structural Foundation,” established a basis for formally describing concepts, types, relationships, responsibilities, and constraints and examining them through purpose-specific views. The Static Model Harness connects those structural descriptions to checking and execution so that constraints continue to apply during AI-assisted generation and modification.

Object models retain formal structure; literate models express intentions and judgments in prose; knowledge models retain semantic relationships and their grounds. Sharing the names, identifiers, and definitions in the BoK glossary maintains correspondence among these three models. Views such as class diagrams select structures from the source model to suit a purpose. The model and glossary remain the authoritative sources of meaning.

1.3 Describing Software Close to Domain Knowledge

Model-driven development describes software with less impedance mismatch against domain knowledge. Directly expressing concepts such as orders, order lines, products, and monetary amounts, together with their relationships and rules, makes it possible to examine business

meaning before expanding it into programming-language implementation details.

In SimpleModeling, Textus directly executes executable models written in CML. AI automatically supplies programming-language implementation where the model alone is insufficient, allowing developers to concentrate on modeling. With this division of work, the model becomes, in effect, the developer's programming language. Supplementary implementation must also obey the model's contracts.

2 Structures Described by Static Models

Representing business objects in software requires defining their identity, shared properties, and relationships with other objects. Static models express these in terms of types, classification, ownership, multiplicity, and related structures.

2.1 Distinguishing Persistent Objects from Values

In SimpleModeling, an entity is a persistent object that continues to exist as the same business object beyond a program's lifecycle. It retains its identity even as its state changes. An order survives the end of an execution and remains the same order when its delivery address changes in a later execution. Describing this identity in the model establishes a criterion for checking generated or modified implementation.

Value objects represent values such as monetary amounts and periods. Objects with the same value are treated as equivalent; whether an individual object continues to exist as the same object is not significant. Describing money as a number paired with a currency makes value equality and the conditions for operations explicit. Type and value constraints belong in the model, with concrete value validation connected to the execution boundaries that need it.

2.2 Classification, Ownership, and Other Relationships

Relationships in the static structure of an object model are organized around classification, structure (ownership), and other relationships. Classification corresponds to is-a relationships, and structure to has-a relationships. Within this overall organization, each relationship specifies roles, multiplicity, and conditions for ownership and lifecycle.

Generalization provides the basis for classification, supplemented by powertypes and traits for expressing classification and commonality. Whole – part structures use aggregation and composition to distinguish sharing from ownership. Other connections between objects specify their roles as associations. Trait composition supplements commonality; its application is not uniformly treated as an is-a relationship.

2.3 Making Classification and Shared Properties Explicit

Generalization is the foundation for expressing classification and shared properties. A more general concept gathers common properties, which more specific concepts refine. For compatibility with programming languages, SimpleModeling uses single inheritance in its generalization hierarchy. Powertypes express classification axes, and traits compose shared partial models, replacing some of the functions otherwise served by multiple inheritance.

A powertype expresses a classification axis, such as customer type or product category, and the categories used on that axis. It supports classification from multiple perspectives alongside the basic single-inheritance hierarchy. The model defines the meaning of each axis, providing a basis for adding categories and designing their implementation.

A trait defines shared features, responsibilities, and constraints across generalization hierarchies as a reusable partial model. Composing that partial model into several classes supplies some of the commonality otherwise expressed through multiple inheritance. Mapping it to a programming-language trait or interface is an implementation policy.

2.4 Understanding Domain Structure through Classification and Whole – Part Relationships

Two basic questions help us understand business objects: “What is it a kind of?” and “What is it made of?” The first reveals commonality and differences; the second reveals the whole formed by its parts. Object modeling gives these questions particular importance as is-a and has-a relationships.

An order, for example, consists of multiple order lines. Treating the order and its lines as one whole lets us reason about operations such as adding a line, changing a quantity, and calculating a total in terms of that structure. Deciding what to treat as one object and what it contains establishes the model’s structural foundation.

The product referenced by an order line plays a different role from a constituent part of the order. Products are managed independently and referenced by many orders. Distinguishing internal parts from independent objects also clarifies responsibility for changes and management.

The Static Model Harness retains these classifications and wholes in the model, providing criteria for preserving their structure during AI-assisted implementation and modification.

2.5 Aggregate Model, View Model, and Display Model

Building on entities and value objects, SimpleModeling provides Aggregate Models, View Models, and Display Models. These explicitly describe how a business whole is composed, how its information is structured for use, and how that information is presented to users.

An Aggregate Model represents related entities and value objects as one whole whose business consistency is maintained. It defines a root and constituent elements, together with the whole’s

s state, operations, and invariants. For an order, the order is the root containing order lines, with quantities and amounts kept consistent with the order total. The model expresses rules and behavior for the whole as well as relationships between individual objects.

A View Model projects information held by entities and other objects into a structure suited to its purpose. It selects the required attributes and relationships to form information appropriate for its consumers while preserving business meaning. An order overview might contain the order number, customer name, and total amount; a detailed view might also include lines and delivery address. Describing correspondence between the source model and each view defines the information required for each use.

A Display Model represents information from a View Model as a structure for presentation to users. It describes lists, details, groups of fields, headings, displayed values, and entry points for operations. For example, an order number can head a list item, the total amount can be a displayed field, and opening the order details can be an available action. Mapping business information to presentation roles lets the model feed the presentation mechanism.

An Aggregate Model captures the order as a business whole. View Models create information structures such as an order overview or order details. Display Models define how those structures are presented as list and detail screens. Explicitly defining the three models and their correspondence supplies criteria for preserving business rules, information meaning, and presentation roles during AI-assisted generation and modification.

2.6 Classifying Entities with Purpose-Specific Stereotypes

Entity objects describe domain objects handled in business. Those objects have structures and behaviors suited to their purposes and properties. SimpleModeling defines these characteristics as stereotypes.

Resource, representing a resource, and Task, representing a unit of processing, are stereotypes that further classify entity objects. Selecting an appropriate stereotype incorporates structure and behavior suited to the object's purpose. Resources describe conditions for ownership, sharing, and use; tasks describe contracts such as inputs, results, and required resources. These descriptions retain information used when choosing execution policies.

The static model defines the resources a task uses and its input/output relationships. The dynamic model describes execution order, waiting, retries, and termination conditions.

2.7 Structural Conditions in an Order Model

Consider an order with an order identifier that owns one or more order lines through composition. Each line references one product. A line belongs to only one order at a time, while a product may be referenced by multiple lines. Money is a value object containing a number and a currency.

In this structure, each line has exactly one owning order. Two orders owning the same line at the same time would violate the constraint. A product, by contrast, can be referenced by

multiple lines. Distinguishing owned parts from references to independent objects makes the order's boundary explicit.

3 Making Models Function as a Harness

Established concepts, components, and patterns provide the modeling framework, and their rules connect to checks and execution in the processing system. Static checks, runtime checks, and semantic review work together to constrain AI-assisted generation and modification and to check correspondence with use cases.

3.1 Integrating Model Checks and Execution into Development

Enterprise application development has accumulated many practical concepts, components, and patterns through decades of software engineering. SimpleModeling provides a foundation for describing them as models and executing those models directly.

CML defines the elements and structure of models through its grammar. The processing system reads the descriptions and performs the applicable checks, while Textus directly executes executable models. Passing AI-generated or modified models and supplementary implementation through this path makes contract violations detectable at development and execution boundaries. This mechanism turns the model-driven development foundation into a harness for AI use.

This framework guides the construction of consistent object models using established concepts and composition methods. AI also describes models and supplies implementation within that framework, reducing the introduction of custom structures or processing where established mechanisms already suffice.

For the harness to enforce constraints, the relevant rules must be represented in the model and connected to checks or execution mechanisms that handle them. Build-time model checks and execution-boundary checks become part of the workflow, with violations resolved before proceeding. Enforcement extends to the rules supported by the connected checking and execution mechanisms.

3.2 Combining Static Checks, Runtime Checks, and AI Review

Conditions that can be established before execution, including model-definition types, reference targets, and relationship declarations, are subject to static checks. Conditions dependent on runtime state—actual values, multiplicity among existing objects, and ownership during concurrent updates—connect to runtime checks and controls. Static and runtime checks divide responsibility according to the nature of each rule.

Reviews by textus-cbd-support apply both to executable models and to models that are not directly executed, supporting build-time checks for contradictions. Results should distinguish mechanically decidable conditions from possible inconsistencies that AI identifies through semantic relationships. Human judgment is also needed to establish whether the model fits business

facts and objectives.

3.3 Checking Correspondence with Use Cases

Alongside human reading of natural-language use cases to guide design and implementation, CML grammar and AI make use cases available for model checking. Reviews by textus-cbd-support can examine correspondence between the concepts, relationships, and responsibilities referenced in scenarios and the static model. Executability and checkability are separate properties.

For a use case that confirms an order, checks include whether the order can be identified, whether relationships between lines and products are defined, and whether money has a shared meaning. If a scenario assumes shared order lines while the static model defines exclusive ownership, the two descriptions need to be reconciled. Requirements traceability helps establish what the described structure is intended to achieve.

4 Using Models in Implementation and Change Management

Identity and ownership described in the static model can guide Textus when saving and restoring objects and defining update boundaries. Maintaining term-based correspondence with documents and knowledge also makes it possible to trace the supplementary implementation, checks, and business explanations that need revision when the model changes.

4.1 Connecting Static Models to Persistent Objects

A key property of static modeling is the ability to handle the described objects as persistent objects. Connecting identity, attributes, relationships, and ownership structure to saving, restoration, and updates allows the same objects to be used across executions.

When a relational database is used, object-relational mapping defines correspondence between objects and tables, attributes and columns, identifiers and keys, and object relationships and references. In SimpleModeling, Textus automates object-relational mapping to connect static model descriptions to persistence. Generalization and ownership also provide a basis for designing this correspondence.

In the order example, the order and its lines are saved and restored while identity is maintained through the order identifier. The Aggregate Model's consistency boundary is related to the persistence update unit. Business rules for history retention, deletion, and concurrent updates are made explicit and implemented within the capabilities of the selected storage method and processing system.

4.2 Connecting Model Semantics to Execution Policies

The classification, identity, ownership, and relationships of model elements also provide executable metadata for the processing system. They can inform the mapping of entity identity to persistence and cache keys, ownership boundaries to update and locking units, and resource/task contracts to scheduling and resource allocation. This requires an explicit correspondence between model semantics and execution policies.

Model identifiers and boundaries can also support distributed placement, concurrency, and observation. Persistence and locking strategies are selected in light of performance, reliability, security, and other requirements and implemented within the capabilities of the processing system. This connection links model meaning to the selection and application of policies.

4.3 Connecting Knowledge and Literate Models through Terms

Terms connect the concepts and relationships of static models to knowledge and literate models. Sharing glossary names, definitions, and identifiers links model elements with documents explaining business meaning and the grounds for decisions. For example, a document can explain what constitutes an order while the same term links to the order's type, attributes, and relationships.

In the knowledge model, RDF nodes corresponding to terms retain semantic relationships between documents and object models. When terms or models change, their connections to documents, structures, and knowledge can be traced and checked for consistency. Actual safeguards are established by integrating tools that check term references and inter-model correspondence into development.

Content expressed in prose within business models, requirements models, and domain knowledge is connected to static models as literate models so that it remains useful throughout development. Building on this connection, the immediate goal is to expose object and literate models represented as knowledge to generative AI through MCP. Part 8 will address this knowledge-delivery environment and the Knowledge Harness.

4.4 Reflecting Model Changes in Implementation and Verification

When changing a static model, confirm the objects' meaning and rules and align the CML descriptions, AI-supplied implementation, and checks. Adding currency to money, for example, affects value equality, addition rules, storage format, and use-case assumptions as well as adding an attribute. Use the model changes to determine the affected scope, then proceed to build-time model checks and runtime verification.

Review examines clarity—whether a description conveys the same meaning to its readers; consistency—whether definitions conflict; validity—whether the content serves its purpose; and accuracy—whether it matches real-world facts and rules. AI supports candidate creation and

inconsistency detection, while humans make business judgments about identity, ownership, and lifecycle.

5 Summary and the Connection to Dynamic Models

The Static Model Harness uses concepts, components, and patterns established through software engineering as a modeling framework that can be checked and executed. It makes types, identity, classification, ownership, relationships, and multiplicity explicit and supports object models built with established composition methods. Passing AI-assisted generation and modification through this framework detects contract violations and limits unnecessary custom structures and processing.

Part 4 will use state machines and workflows to describe which events the defined objects receive and under what conditions they change state. Part 5 onward will proceed toward architecture, programming, and runtime realization. Building on glossary-mediated semantic connections among models, Part 8 will address the Knowledge Harness for providing shared and project-specific knowledge to generative AI.