Erfindung Generische Modelle

Unsere Erfindung besteht im Kern aus einem reflektierten Typsystem für Entitäten von Modellen, das wir zunächst als Erweiterung für Java und auch TypeScript entwickelt haben. Die Erfindung erfolgte als konsequente Entwicklung mit Hinsicht auf die Bedürfnisse in modernen Softwaresystemen und verwertet Erfahrungen aus mehreren Dekaden. Das Ergebnis ist eine Konsolidierung, Formalisierung und Raffinierung von typischen Praktiken, welche die Community in unausgereifter Form ständig einsetzt. Als Vergleich bietet sich das Verhältnis der pseudo-objektorientierten Programmierung in C mit structs zur Erfindung und Verwendung einer dedizierten objektorientierten Sprache wie C++ an. Ein ähnliches Beispiel ist das Verhältnis von mühsam konstruierten SmartPointer Lösungen in C++ zu den natürlichen Referenzen in Java und ähnlichen Sprachen. Die dedizierte Erfindung setzt jeweils die wesentlichen Leitplanken für das Paradigma schon zu Beginn und macht die mühsame Kür zur eleganten Pflicht.

Die Erfindung wurde zunächst als proprietäres Gut entwickelt. In 2023 wurde sie dann unter der LGPLv3 Lizenz vollständig Open Source gestellt und wird auf GitHub (https://github.com/orgs/hiconic-os) gepflegt. Der Open Source Code ist sehr umfangreich und reicht von grundlegenden Spracherweiterungen, Tooling und Abhängigkeitsmanagement bis hin zu einer Anwendungsplattform und ihren Standardmodulen. Was im proprietären Rahmen erreicht wurde, ist beachtlich, es muss aber auch ehrlich eingestanden werden, dass die kommerziellen Bedingungen dazu beigetragen haben, dass die Aufbereitung für eine Community (vor allem Dokumentation) rückständig ist.

Mit dem Verlassen des proprietären Rahmens entsteht nun die Frage, wie das Projekt weitergetrieben und versorgt werden kann. Zum einen könnte eine Integration in ein bestehendes Laufzeitsystem wie quarkus oder spring-boot geschehen und zu anderen könnte eine native Einbettung in eine moderne Programmiersprache stattfinden, um das volle Potential zu entfalten.

Einführung in das neuartige Typsystem für Modelle

Das Typsystem mit multipler Vererbung basiert auf Interface-Definitionen, welche Entitäten mit typisierten Eigenschaften beschreiben. In Java sind die Eigenschaften mangels eines entsprechenden Sprachfeatures durch ein Paar von Getter und Setter abgebildet. Die Eigenschaften haben jeweils einen Typ mit den folgenden Möglichkeiten:

- *Elementartypen*
  - *Primitive Typen*
    - **boolean**
    - **integer**
    - **long**
    - **float**
    - **double**
    - **decimal**
    - **date**
  - *Benutzerdefinierte Typen*
    - **entity**
    - **enum**
- *Generische Kollektionstypen für Elementartypen*

- **set**
- **list**
- **map**

Das Typsystem soll im Allgemeinen die Kapselung von objektorientierter Programmierung hinter sich lassen und stattdessen die Trennung von Daten und Code fördern. Es gibt wenige Ausnahmen, in denen default Methoden auf Entitätentypen eine reduzierte Kapselung benutzen. Die Gründe dafür sind Performance und Eleganz.

Die Normalisierung von Daten und Code im Sinne von Homoikonizität soll diese Trennung von Daten und Code auf einer höheren Ebene überwinden und eher in Richtung funktionale Programmierung für Modelle zielen.

Compiler für Entitäten Interfaces

Durch die Nutzung von Interfaces für die Definition von Entitäten erfolgt die Implementation automatisch durch einen Compiler. Während Interfaces durch Mehrfachvererbung den Gestaltungsraum des Entwicklers erweitern, nimmt die automatische Implementation gleichzeitig redundante Arbeit ab. Es ergeben sich durch die automatische Implementation darüber hinaus weitere wesentliche Vorteile und Leistungen. Dazu gehört die detaillierte und hocheffiziente Reflektion. Jeder benutzerdefinierte Typ erhält automatisch eine reflektorische Instanz von EntityType, die über den Namen T im Namensraum des Interfaces als Literal erreichbar ist. Das T Literal ist als Zugang zur Reflektion mit dem class Literal der Java Reflection vergleichbar. Über diese Instanz können die Vererbung erkundet, Eigenschaften gelistet und Instanzen erzeugt werden. Die Reflektion der Eigenschaften erlaubt wiederum den Typ der Eigenschaft zu erfahren und ermöglicht generischen Lese- und Schreibzugriff für konkrete Eigenschaftswerte auf zugehörigen Entitäten. Im Gegensatz zur bekannten Java-Reflection ist der generische Zugang über die kompilierte Reflektion für Entitätstypen eleganter und mit keinerlei Geschwindigkeitseinbußen im Vergleich zu einem direkten ausdrücklichen Zugriff über Setter oder Getter verbunden. Auf Basis der damit möglichen Generik können effiziente domänenunspezifische Algorithmen geschrieben werden, wie sie für aktuelle IT-Bedürfnisse dringend notwendig sind. Ein griffiges Beispiel ist etwa Serialisierung in generische Strukturformate wie JSON oder YAML. Ein weiterer Vorteil der automatischen Implementation der Entitätentypen ist die Unterstützung von Property Access Interceptors. Diese können generell in den Lese- und Schreibzugriff auf Eigenschaften geschaltet werden. Auf diese Weise werden Cross-Cutting Concerns wie Lazy Loading und Manipulation Tracking abgebildet. Eine besondere Leistung der automatischen Implementation ist, dass die Felder der Eigenschaften auch in der Lage sind, alternative evaluierbare Entitäten anstelle der eigentlichen typsicheren Werte zu halten. Ein Property Access Interceptor kann in diesem Falle die dynamische Evaluation mit dem Ergebnis des eigentlichen Wertes vornehmen und somit den typsicheren Getter beliefern.

Im Falle der Lösung für Java, arbeitet der Compiler zur Laufzeit auf Basis von Java-Bytecode Generierung. Im Falle von Typescript haben wir zur Wiederverwendung unserer Core-Implementationen der Java-Lösung den GWT Transpiler eingesetzt, um zunächst eine reine Javascript Bibliothek zu erhalten. Hierbei werden GWT-Codegeneratoren zur Kompilierzeit eingesetzt, um die Entitäten Interfaces zu implementieren. Es werden aber auch Javascript Codes erzeugt, um die Interfaces und ihre Implementation auch zur Laufzeit zu erzeugen. Der GWT Transpiler erzeugt standardmäßig kompaktes, verschleiertes Javascript mit der Ausnahme von Namensräumen öffentlicher APIs. Um das Ergebnis des GWT Transpilers für unsere öffentlichen APIs und die Entitätentypen auf das Level Typsicherheit aufzuwerten, haben wir einen weiteren Generator zur Erzeugung von zugehörigen Typescript Deklarationen geschrieben.

Modelle für alle Zwecke und Homoikonizität

Die Modelle, die mit den Entitätentypen des Typsystems modelliert werden, sind ausdrücklich für alle Zwecke gedacht und bilden damit eine Verallgemeinerung zu den vielen bestehenden Modellsystemen. Letztere sind leider einseitig an spezielle Domänen wie Serialisierung, RPC, Web API und Persistenz gebunden und bringen daher oft Übersetzungsaufwände und eingeschränkte Leistungen mit sich. Gleichwohl werden von unserem verallgemeinerten Ansatz die isolierten Anwendungsfälle der bestehenden Modellsysteme bedient. Im Sinne eines ganzheitlichen Systems werden mit besonderer Sorgfalt auch solche Domänen bedient, welche die Homoikonizität des Systems fördern sollen. Dazu gehören reflektorische, konfigurative und funktional evaluierbare Modelle mit angebundenen Expertensystemen wie die folgenden Beispiele zeigen:

Model	**Kategorisierung**	**Beschreibung**
meta-model	reflektorisch	Kann alle Modelle inklusive sich selbst beschreiben und dient als Eingabe für Compiler und Transpiler sowie als Ein- und Ausgabe von Modeltransformern für UML, XSD, Open Api Schema, GraphQL Schema
meta-data-model	konfigurativ	Basismodell für Metadaten wie sie an Elemente des meta-model zur Beschreibung, Persistenzanbindung, Expertenbindung, Einbringung von Cross-Cutting Concerns usw. angebracht werden können.
value-descriptor-model	funktional evaluierbar	Basismodell für evaluierbare Typen, die als dynamische Ersatzwerte in Entitäteneigenschaften verwendet werden können.
manipulation-model	funktional evaluierbar	Beschreibt Instanzierungen und Veränderungen von individuellen Entitäten, um diese Vorgänge für Persistenz, Undo/Redo und MVC reproduzierbar zu machen. Stützt somit auch event-source Datenbanken.
deployment-model	reflektorisch	konfigurativ
service-api-model	funktional evaluierbar	Basismodell für Service Requests, die zum Beispiel in APIs für CLI, Web oder Queues zur Anwendung kommen können.
query-model	funktional evaluierbar	Modell zur Abfrage von Persistenz in OO oder SQL Datenbanken.

Gerade die konsequente Verwendung des neuen Typsystems für grundlegende Modelle zur Förderung von Homoikonizität macht seine besondere Bedeutung für die Normalisierung von Softwaresystemen deutlich. Durch diese Normalisierung wird eine wesentliche Reduktion von Redundanz und Software-Entropie möglich. Das Zusammenspiel der homoikonen Modelle und ihrer Expertensystem untereinander und die Möglichkeit sie auch auf sich selbst anzuwenden, ist eine effektive Grundlage für reflektierte, (selbst)instrumentierbare, portable und verteilte Systeme. Die vielen vorbereiteten generischen Algorithmen auf Basis der Reflektion tragen ebenfalls dazu bei, das System logisch abstrakt und portierbar zu halten.

Die pivotale Positionierung der Modelle

Das aus dem Typsystem entstandene Paradigma wendet eine “Modelle zuerst” Strategie an. Das bedeutet, dass im Gegensatz zu vielen anderen Systemen zunächst keine Grammatik für bedeutungsvolle Domänen definiert wird, sondern immer erst Modelle. Die Modelle sind damit pivotal. Komfortable und ausdrucksstarke Grammatiken können nach Bedarf danach hinzukommen. Ein Parser für eine solche zusätzliche Grammatik erzeugt immer Instanzen des zugehörigen Modells. Diese können dann wiederum mit den passenden Experten weiterverarbeitet werden. Auf diese Weise kann es auch unterschiedliche Grammatiken mit unterschiedlichem Brennpunkt zu ein und demselben Modell geben.

Anwendungsfall Objektorientierte In-Memory Datenbank

Da sich reflektierte Entitäten ideal für generische Suchen und Veränderungen anbieten, wurde eine In-Memory OO Datenbank auf Basis des Typsystems, des query-model und des manipulation-model entwickelt. Zusammen mit dem zugehörigen session-api kann auf diese Persistenz gleichartig zugegriffen werden wie auf eine alternative SQL Datenbank.

Anwendungsfall Verteilte modulare Anwendungsplattform

Mit Hilfe des homoikonen Ansatzes wurde eine Anwendungsplattform für verteilte Systeme entwickelt. Die Plattform verwaltet und reflektiert Modelle und konfigurierbare Systemkomponenten aus Modulen in ihrer Cortex-Persistenz. Als wesentliche Basismodelle der Cortex-Persistenz dienen das meta-model, das meta-data-model, das deployment-model und das service-api-model. Durch die Cortex-Persistenz können modellierte funktionale Systemkomponenten und Modelle miteinander vernetzt und in Arbeit genommen werden. Die dabei entstehenden System- und Anwendungsfunktionen werden über Erweiterungen des service-api-model logisch aufrufbar. Über verschiedene automatische Endpunkte für Web/HTTP, Message-Queues und CLIs wird der Aufruf durch entfernte Komponenten (andere Prozesse und Maschinen) ermöglicht.

Die Anwendungsplattform kann durch benutzerdefinierte Modelle und Modulen mit Hilfe von Abhängigkeitsverwaltung erweitert werden. Die Plattform kann dadurch beliebige generische oder fachliche Funktionalität im Verbund beheimaten und bereitstellen. Jeder kann Module und Modelle für diese Erweiterung entwickeln. Die Plattform lässt damit den typischen Siloeffekt hinter sich.