



„Low Code“ mit DSLs

Filial-Warenwirtschaftssystem auf Basis von Java und drei DSLs in JetBrains MPS

Daniel Stieger, Wolfgang Messner, Martin Mölk, Bernd Flunger

Domain-Specific Languages (DSLs) und Language Workbenches sind in der Praxis noch nicht wirklich angekommen. Wir berichten von unseren Erfahrungen mit diesen Werkzeugen und einem Projekt, das wir vor 12 Jahren begonnen haben. Wir beginnen mit der Ausgangssituation unserer Reise, zeigen auszugsweise verwendete DSLs und ziehen ein Projekt-Resümee.

MPREIS, einer der größten Lebensmitteleinzelhändler in Österreich, beauftragte uns, die Modellwerkstatt, vor 12 Jahren mit der Entwicklung eines Filial-Warenwirtschaftssystems. Die primäre Aufgabe des Systems sollte die Unterstützung der wesentlichen Geschäftsvorfälle wie Wareneingang, Warenausgang oder Abschreibungen durch Verderb, Schwund usw. von Waren in einer Filiale sein. Daraus lässt sich dann wert- und mengenmäßig ein Warenbestand führen, wenn das System um umfangreiche Inventur-Funktionalitäten ergänzt wird. Ferner sind aber auch Geschäftsprozesse, wie die manuelle Disposition, also die Bestellung von Waren, vollumfänglich zu unterstützen.

Die Ausgangssituation

Mit einer Eigenentwicklung wollte man die hauseigenen, spezifischen Anforderungen besonders berücksichtigen. In Zukunft sollte man schnell und dynamisch auf Änderungen reagieren und das Experimentieren mit innovativen Ideen in der Filial-Bewirtschaftung unterstützen können, „Handel ist Wandel“ lautet schließlich die älteste aller Binsenweisheiten des Handels. Die Eigenentwicklung sollte den filialisierten Handel mindestens für die nächsten 30 Jahre stärken, hohe Flexibilität sicherstellen und Veränderungen in Abläufen schnell und effizient abbilden.

In der Diskussion mit dem Management hatten sich damals auch Randbedingungen abgezeichnet. Eine Orientierung an modernen Open-Source-Technologien wurde gewünscht. Grundsätzlich nahm das Management Java als langfristig stabiles Ökosystem wahr. Große Skepsis herrschte allerdings gegenüber schwergewichtigen Applikationsservern, verbunden mit Schlagwörtern wie J2EE oder Hibernate. Entwickler mögen sich auf die Fachaufgaben und Funktionalitäten konzentrieren. Sie sollten nicht durch einen unnötig komplexen Technologie-Stack aufgehalten werden. Zudem war zu diesem Zeitpunkt auch der Begriff der „Technischen Schulden“ bei Eigenentwicklungen intern präsent.

Der Begriff ist eine von Ward Cunningham bereits 1992 geprägte Metapher [Cun92], die ausdrückt, dass bei Softwaresystemen der Wartungs- und Erweiterungsaufwand durch suboptimale Entscheidungen und Architektur-Erosion über die Zeit hinweg zunimmt. Angesprochen sind wartungsintensive Softwaresysteme, die „historisch gewachsen“ sind – man kennt das zur Genüge. MPREIS und die Modellwerkstatt hatten daher sehr früh zusammen drei langfristige Herausforderungen identifiziert, die wir für das Warenwirtschaftssystem berücksichtigen wollten:

- *Klare und verständliche Architektur*, die die langfristige Wartbarkeit und Anpassungsfähigkeit des Systems sicherstellt und den technischen Schulden entgegenwirkt. Klar definierte Muster und Abstraktionen sollten den Schwerpunkt vor allem auf die fachlichen Funktionalitäten legen, verwendete technische Frameworks sollten hingegen austauschbar sein. Die fachliche Geschäftslogik dürfe keinesfalls mit technischen Komponenten verwoben werden. Wir strebten im Hinblick darauf eine möglichst eindeutige Trennung an.

- Aufgrund der beschränkten Implementierungskapazitäten griffen wir auch das Schlagwort *Produktivität* auf. Entwickler sollten sich vor allem auf die Komplexität der Fachdomäne fokussieren, die technischen Details durften nicht im Vordergrund stehen. Modellgetriebene Entwicklung mit vorgefertigten Softwarekomponenten waren uns bereits bekannt. Zudem hatten wir bereits positive Erfahrungen mit generiertem Code aus einem Projekt im Maschinenbau. Sicher müsste sich auch aus fachlichen Datenmodellen eines Warenwirtschaftssystems ausführbarer Code generieren lassen, um die Produktivität bei der Implementierung zu erhöhen.
- Für die Umsetzung wollten wir ein *breiteres Publikum* ansprechen, nicht nur Hardcore-Java-Entwickler. Langfristig sollte es für MPREIS möglich sein, interessierte Personen mit Programmierverständnis an der Weiterentwicklung zu beteiligen. Auch in der Vergangenheit waren im Unternehmen neben Informatikern viele Autodidakten an der Implementierung von Lösungen beteiligt. Die Arbeitsmarktsituation mit dem Fachkräftemangel in Österreich schärfte bei diesem Punkt auch nochmals das Bewusstsein.

Mit welchen Konzepten kann man also eine verständliche Architektur schaffen, die langfristige Wartbarkeit, aber auch die Produktivität bei der Implementierung erhöhen? Wie kann man dabei Leute einbeziehen, die sich für die fachliche Domäne interessieren, jetzt aber nicht Java-Experten sind? Zum einen sollten wir einen zukunftsweisenden Technologie-Stack vorschlagen, zum anderen dann direkt den Startschuss zur Implementierung der eigentlichen Applikation – das Warenwirtschaftssystem für ca. 300 Filialen – setzen.

DSLs to the rescue

Domänenspezifische Sprachen sind spezielle, anwendungsspezifische Computersprachen, die darauf zugeschnitten sind, Probleme eines bestimmten Fachbereichs (der sogenannten Domäne) auszudrücken und zu lösen. Sie sind also speziell für den Einsatz

in einem bestimmten, fachlichen Bereich konzipiert und nicht universell anwendbar. Die Ideen hinter DSLs sind in der Literatur hinlänglich diskutiert (z. B. [Fow11, Völ13]). Auch in JavaSPEKTRUM ist Anfang letzten Jahres ein guter Artikel zu DSLs erschienen [Schä22]. Während General Purpose Languages (GPLs) wie Java oder Python eher technisch orientiert sind und dadurch für alle Aufgabenstellungen angewendet werden, konzentrieren sich DSLs lediglich auf einen konkreten fachlichen Anwendungskontext. Ziel ist es, die Terminologie und Notation dieses Anwendungskontexts abzubilden und geeignete fachliche Abstraktionen zu schaffen, sodass Anforderungen möglichst leicht verständlich formuliert werden können. Diese „fachlich getriebenen Programme“ müssen sich später natürlich zu ausführbarem Code transformieren lassen.

Theoretisch versprechen DSLs, die Produktivität bei der Erstellung von Lösungen zu erhöhen, da schlichtweg die Anzahl an Programmzeilen, die gelesen, geschrieben und verstanden werden müssen, sinkt. Gerade eine ordentliche Syntax erhöht die Lesbarkeit deutlich. Auch die Qualität steigt, wenn durch klare Architekturvorgaben einer DSL die Konformität im Projekt gestärkt wird. Für die Entwickler sinkt die (unnötige) Anzahl an Freiheitsgraden gegenüber einer GPL, was Fehlern entgegenwirkt. DSLs erlauben eine umfangreiche Validierung und Verifikation von Inhalten, da sie meist semantisch ausdrucksstärker sind, wenn sie wesentliche Konzepte einer Domäne direkt abbilden. Nicht zuletzt sind DSLs aber gerade für die Kommunikation im Entwicklerteam und mit den Fachexperten im jeweiligen Bereich äußerst hilfreich. Man muss erst die wesentlichen Konzepte aus einem fachlichen Anwendungskontext heraus exakt definieren und benennen, um sie in einer DSL abbilden zu können. Genau diese zentralen Konzepte sind es dann, die in der Kommunikation wieder und wieder verwendet werden. Man hat also einen Namen für die „Dinge“.

Im Zusammenhang mit DSLs wird auch immer gerne zwischen:

- einem fachlichen Aspekt und
 - einem technischen Aspekt
- unterschieden. Die Vorteile der Trennung von fachlichen und

technischen Aspekten sind ebenfalls hinlänglich bekannt (z. B. [Völ04, Lil20]). Während Geschäftsregeln, Validierungen, Berechnungen oder Buchungen fachlichen Bausteinen zuzuordnen sind, sollen technische Bausteine ausschließlich technische Anforderungen lösen. Dazu zählen beispielsweise der Datenbank-Anschluss oder die Benutzeroberflächen (UIs), also die technische Basisplattform für fachliche Bausteine. Minimiert man Bausteine, die eine Mischung von fachlicher Anforderung (Fachlichkeit) und rein technischen Belangen darstellen, entsteht ein gut modularisiertes System. Die konkrete Implementierungstechnik lässt sich idealerweise sogar hinter der Fachlichkeit verstecken, sodass sie für den „fachlichen Entwickler“ an Relevanz verliert.

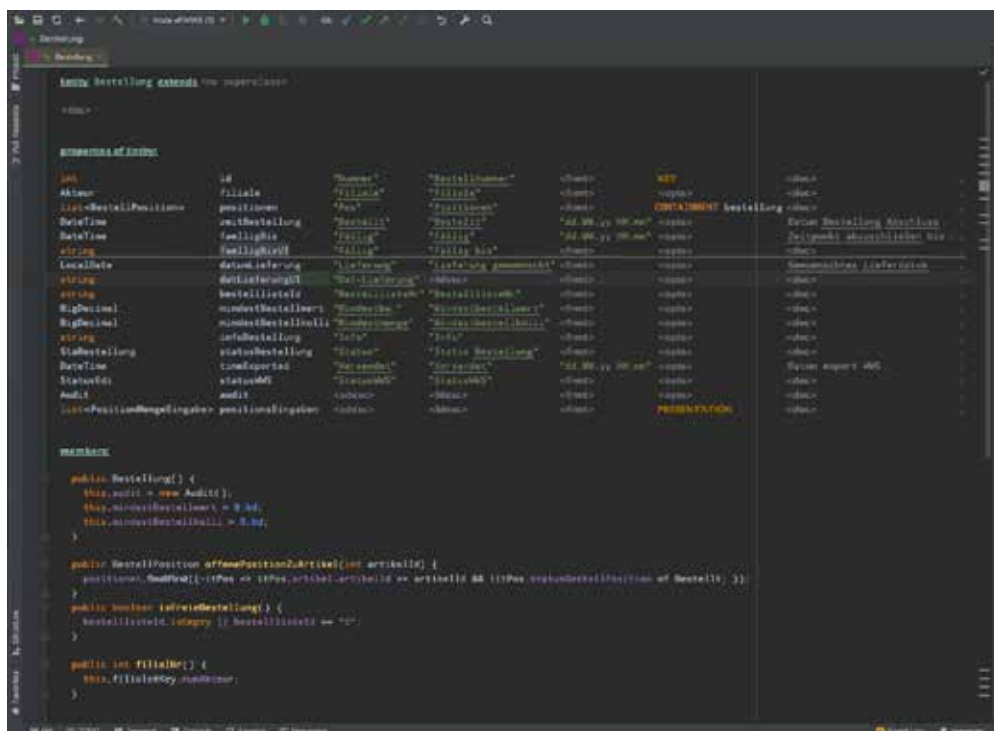


Abb. 1: Entität Bestellung im MPS



Dr. Daniel Stieger ist seit 2010 bei der modellwerkstatt.org und befasst sich dort mit DSL-Design und notwendigen Laufzeitumgebungen. In der Sprachentwicklung konzentriert er sich vor allem auf JetBrains MPS. E-Mail: daniel.stieger@modellwerkstatt.org



DI Wolfgang Messner kommt ursprünglich aus dem Maschinenbau und beschäftigt sich bei der modellwerkstatt.org nun stark mit Geschäftsanwendungen & Geschäftsprozessen. Er hilft Unternehmen bei der Applikationsentwicklung und verwendet dabei die eigenen DSLs der modellwerkstatt. E-Mail: wolfgang.messner@modellwerkstatt.org



Martin Mölk war bis 2022 IT-Leiter bei MPREIS und programmiert auch selbst gerne, vor allem im Bereich der Warenwirtschaft bei MPREIS. Innovative Ansätze bei der Eigenentwicklung von Software, verschiedene Programmiersprachen, DSLs und Tools gehören zu seinen Leidenschaften. E-Mail: martin.moelk@mpreis.at



DI Bernd Flunger ist seit 1998 bei mpreis.at und beschäftigt sich seither mit Handelsprozessen, Applikationsentwicklung und dem Aufbau der IT-Infrastruktur für die MPREIS Warenvertriebs GmbH. E-Mail: bernd.flunger@mpreis.at

Für das Projekt Warenwirtschaftssystem haben wir angesichts der eingangs formulierten Herausforderungen dann direkt das Arbeiten mit einer DSL vorgeschlagen. Die Trennung von Fachlichkeit und Technologie wollten wir noch weiter treiben. Durch eine geeignete DSL sollen technische Aspekte überhaupt in den Hintergrund gedrängt werden. Die DSL sollte eine maximale Konzentration auf die fachlichen Anforderungen des Warenwirtschaftssystems ermöglichen. Im Idealfall sollten die Entwickler von technischen Notwendigkeiten und der technischen Basisplattform erst gar nicht abgelenkt werden. Wir strebten an, sie durch eine DSL zu entlasten. Wie eine DSL für ein Warenwirtschaftssystem jedoch in der Praxis konkret aussehen könnte, war bis dato unklar.

Erste Schritte – DDD

Die Domäne eines Warenwirtschaftssystems abzugrenzen, fällt relativ schwer: Prozessabläufe, Dokumente, Geschäftsregeln, Buchungskonten und Warenbewegungen, Stammdaten, aber auch Benutzeroberflächen sind zu spezifizieren. Wo den Fokus setzen, mit

was anfangen? In den ersten Tagen haben wir uns daher an bestehenden Open-Source-ERP-Systemen und deren Meta-Modellen orientiert (beispielsweise ERP5 [ERP5] oder OpenERP [Odo]). Die Meta-Modelle konnten uns dann aber nicht vollends überzeugen. Wir erkannten für unseren Anwendungskontext keine besonderen Stärken.

Zielführender erschien es uns daher, die Anwendungsentwicklung durch eine geeignete Mustersprache auf ein höheres abstraktes Niveau zu heben und mit eigenen Meta-Modellen zu experimentieren. Auf diese Weise erhielten wir im frühen Projektstadium viel Flexibilität. Musterelemente, wie sie im taktischen Domain Driven Design ausdefiniert sind (z. B. [Eva04, Ver13, Kho21]), haben uns als erste Abstraktionsebene überzeugt.

Auf Klassenebene definiert das sogenannte taktische DDD einen Satz von Konzepten, deren Verantwortlichkeiten und wie diese Konzepte miteinander interagieren dürfen. Bekannt sind die:

- *Value Objects*, die keine eigene Identität bilden, deren Wert ausschließlich relevant ist.
- *Entitäten*, als Kernobjekte einer Fachdomäne mit Identität und klar definiertem Lebenszyklus.
- *Aggregate*, die vernetzte Entitäten kapseln und wesentliche Bedingungen in Bezug auf Änderungen sicherstellen.
- *Services*, die fachliche Operationen, Abläufe oder Prozesse der Domäne implementieren, also mit Entitäten und Value Objects arbeiten.
- *Factories* als logisches Konzept, das Entitäten, Value Objects und eben Aggregate erzeugt.
- *Repositories*, die den Persistenzmechanismus für Entitäten und Aggregate kapseln, sodass eine klare Schnittstelle entsteht und sich diese „technische Aufgabe“ nicht in die fachliche Domäne zerstreut.

Die DSL soll sich also stark an die Muster und Konzepte des DDD (Domain-Driven Design) anlehnen. Als zentrales Tool zur Implementierung der DSL und des Warenwirtschaftssystems haben wir 2010 JetBrains MPS ausgemacht. JetBrains MPS ist eine sogenannte Language Workbench zur Implementierung und Anwendung von DSLs. In JetBrains MPS wird die DSL sowohl definiert als auch angewendet. Entwickler können die definierte DSL dann gleich im MPS (Meta Programming System) zum eigentlichen Schreiben von Programmen nutzen. Der Artikel [Schä22] diskutiert ebenfalls JetBrains MPS als Language Workbench.

Gegenüber anderen Werkzeugen überzeugte uns vor allem der projektionale Editor und die damit verbundenen Möglichkeiten, allen voran das Arbeiten mit mehreren DSLs gleichzeitig. Aufgrund des projektionalen Editors von MPS, bei dem man direkt auf dem Abstract-Syntax-Tree arbeitet, gibt es keine Einschränkungen bei der DSL-Syntax, wodurch sich auch mehrere DSLs aufeinander aufbauen lassen (eine sehr gute Übersicht gibt [JetB]). Und von Haus aus lieferte MPS schon zwei besonders interessante DSLs für uns.

Zum einen verfügt die Language Workbench schon über eine vollständig implementierte und stabile Java DSL. Mit der `jetbrains.mps.baselanguage` wurde die komplette Java-Sprache als DSL im MPS abgebildet. In seiner eigenen DSL kann man sich daher dieser `baselanguage` vollständig bedienen oder einfach nur Konzepte davon verwenden. Letzteres schien uns eine effektive Vorgehensweise. Die oben beschriebenen Konzepte aus dem taktischen DDD haben wir in einer eigenen DSL implementiert, grundlegende Konzepte, allen voran Expressions, das Typsystem oder grundsätzliche Kontrollstrukturen, wie `if` oder `for`, haben wir hingegen der `baselanguage` entnommen.

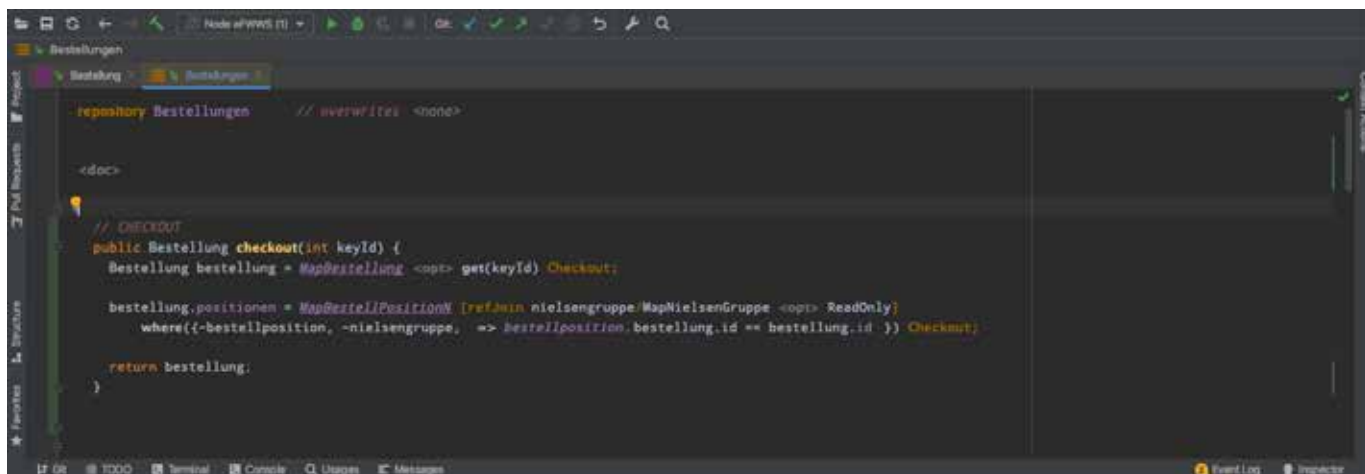


Abb. 2: Laden des Aggregates Bestellung

Zum anderen verfügt JetBrains MPS auch über Erweiterungen zur Java baselanguage, also ergänzende DSLs, die wiederum auf der baselanguage aufbauen. Schon 2010 war eine umfassende Closure- (jetbrains.mps.baselang.closure) und Collections-Unterstützung (jetbrains.mps.baselang.collections) verfügbar! Die Collection DSL baut dabei auf der Closure DSL auf – die wiederum auf der Java DSL aufbaut – und erlaubt das Arbeiten auf Sammlungen in einer angenehmen, funktionalen Syntax. Wir konnten uns also existierender DSLs in JetBrains MPS bedienen. Neben der Java-Grundsprache haben wir beide Erweiterungs-Sprachen in unseren DSL-Stack übernommen. Nachfolgend wollen wir zwei konkrete Beispiele im MPS zeigen: eine Entität und ein Repository.

Entitäten – rein fachlich

Mithilfe von Domänenmodellen wird im DDD komplexe Geschäftslogik abgebildet. Die Domänenmodelle sind dabei Objektmodelle einer Fachdomäne, die sowohl Verhalten als auch Daten abbilden. Dabei soll die Terminologie der Domänenmodelle eben genau der Terminologie der Domäne folgen (Ubiquitous Language). Bausteine der Domänenmodelle sind neben Services, Aggregaten und Value Objects auch Entitäten. Eine Entität benötigt immer eine eindeutige Identifikation, einen Schlüssel. Anhand dieses Schlüssels werden Entitäten unterschieden – das ist bei Value Objects ja genau anders. Von der Erstellung einer Entität bis zur Löschung folgen die Objekte meist einem genau definierten Lebenszyklus.

In Abbildung 1 sieht man die Definition der Entität Bestellung in der MPS IDE. Wie zu erkennen ist, konzentriert sich die Darstellung ausschließlich auf fachlich relevante Aspekte. Keine technischen Java-Annotationen, beispielsweise JPA-Persistenz-Annotationen, stören den Lesefluss. Im Gegenteil: Die tabellarische Darstellung unterstützt das gedankliche Erfassen der zentralen Bestandteile der Entität. Der Entwickler kann auch UI Labels angeben, für den Fall, dass kein eigenes Präsentationsmodell notwendig ist, die Entität also selbst direkt am UI bearbeitet wird. Das ist häufig der Fall. Abbildung 1 verdeutlicht aber auch die Flexibilität, die wir zulassen. Der Entwickler kann fachlichen Java-Code in Methoden direkt der Entität zuordnen. Infrastruktur-Zugriffe sind an dieser Stelle nicht erlaubt – und auch nicht möglich! Hier gibt es klare Einschränkungen, die unsere DSL vornimmt.

Aus der fachlichen Entität erzeugen wir mithilfe eines Code-Generators eine Java-Klasse. Dabei wird der Code-Generator neben

der DSL selbst ebenfalls direkt in MPS implementiert. Den generierten Java-Code möchten wir an dieser Stelle allerdings nicht zeigen. Er ist deutlich umfassender. Wir generieren Getter & Setter, einiges an Boilerplate-Code für einen Entity-Dirty-Mechanismus und zusätzlichen, frameworkspezifischen Glue-Code. Es ist genau dieser Code, der abschreckt und nicht zum Lesen und „Verstehen wollen“ dieser zentralen Entität motiviert.

Persistierung – aber transparent

Repositories bilden im taktischen DDD die Schnittstelle zu Datenbanken und anderen Persistenz-Technologien. Sie laden ein Aggregat anhand einer Spezifikation, also einer spezifischen Abfrage, oder aber einfach anhand des Schlüssels einer Entität – typischerweise der sogenannten Wurzel-Entität des Aggregates. Wir haben für Repository-Abfragen im MPS eine eigene DSL entwickelt, die sich analog zu einer Java Collection verhält. Dabei haben wir uns syntaktisch an die bereits angesprochene MPS Collection DSL (jetbrains.mps.baselang.collections) angelehnt.

In Abbildung 2 wird gezeigt, wie mit der DSL eine Bestellung anhand ihres Schlüssels geladen wird. Erst erfolgt eine Datenbankabfrage, die eine Bestellung anhand des Schlüssels keyId über die MapBestellung lädt. Das ist das Mapping zwischen Entität und Tabelle. Dann werden die Positionen der Bestellung über die MapBestellPositionN geladen und auch gleich der Entität Bestellung angehängt. Im Hintergrund wird mit einer SQL-Datenbank gearbeitet – ein Implementierungsdetail. Das where-Element wird von unserem Code-Generator in ein SQL-Where-Statement übersetzt. Im MPS haben wir bei diesem Where-Element unserer DSL aber eine typensichere Notation in moderner Closure-Syntax mit voller IDE-Code-Completion-Unterstützung! Auch komplexere Ladevorgänge können mit der Mächtigkeit der DSL ausgedrückt werden. So wird in diesem Beispiel die Entität NielsenGruppe durch einen Join-Vorgang geladen und die entsprechende Referenz in der Bestellposition selbst gleich auf die richtige NielsenGruppe gesetzt. Der Anwendungsentwickler kann mit dieser DSL höchst komfortabel auf den Datenbestand zugreifen und die schnellste Ladestrategie formulieren. Der Code-Generator erzeugt die technische Implementierung.

Nächste Schritte – die Präsentationsschicht

Um die fachlichen Operationen und Abläufe des Warenwirtschaftssystems zu modellieren, haben wir also auf die Abstrak-

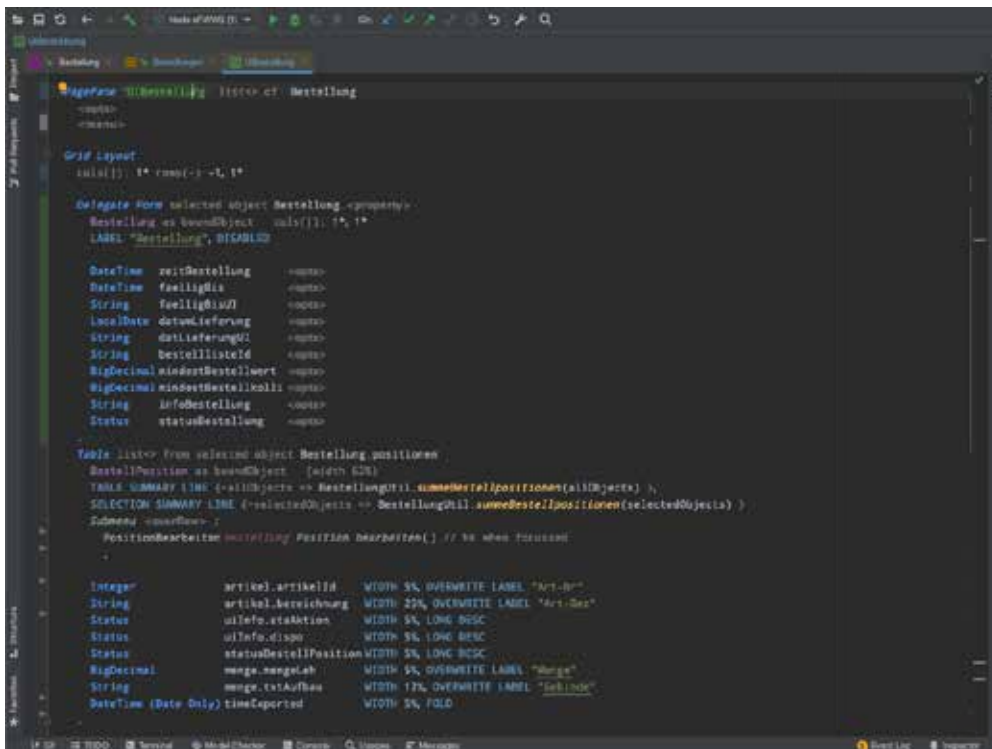


Abb. 3: Einfaches Modell eines Bestellung-UI

tionen von DDD zurückgegriffen. Für die Formulierung der Operationen mussten wir keine eigene DSL entwickeln. Hier haben wir auf die JetBrains Java-Baselanguage und deren Erweiterungen gesetzt. Damit lässt sich Geschäftslogik rund um Aggregate, Entities und Services ausdrücken. Wir können die Fachlichkeit des

det: der dataUx-Sprache.

In Abbildung 3 ist auszugsweise das Standard-UI einer Bestellung modelliert – technologieunabhängig! In einem Grid-Layout werden 7 Felder der Bestellung zweispaltig in einem Formular (disabled) angezeigt. Das Formular ist in einer Zeile des Grids

Warenwirtschaftssystems unabhängig von der verwendeten Basisplattform in Java ausdrücken. Es fehlte jetzt allerdings noch ein wesentliches Bauteil für ein Warenwirtschaftssystem: die Präsentation.

Für eine logische Abstraktion in der Präsentation haben wir uns an bestehenden UIs von Geschäftsanwendungen orientiert. Sehr typisch sind dort Formulare, Tabellen und Reiter. Häufig findet in der einen oder anderen Form das Master-Detail-Pattern Anwendung. Dieses Pattern ist leicht verständlich und bedarf keiner großen Erklärung für den Endanwender. Wenn der Endanwender ein Objekt im Master aus einer Liste auswählt, sieht er die Informationen und alle Aktionen, die für dieses Objekt relevant sind, im Detail. Wir haben diese Ideen mit einer eigenen DSL abgebil-

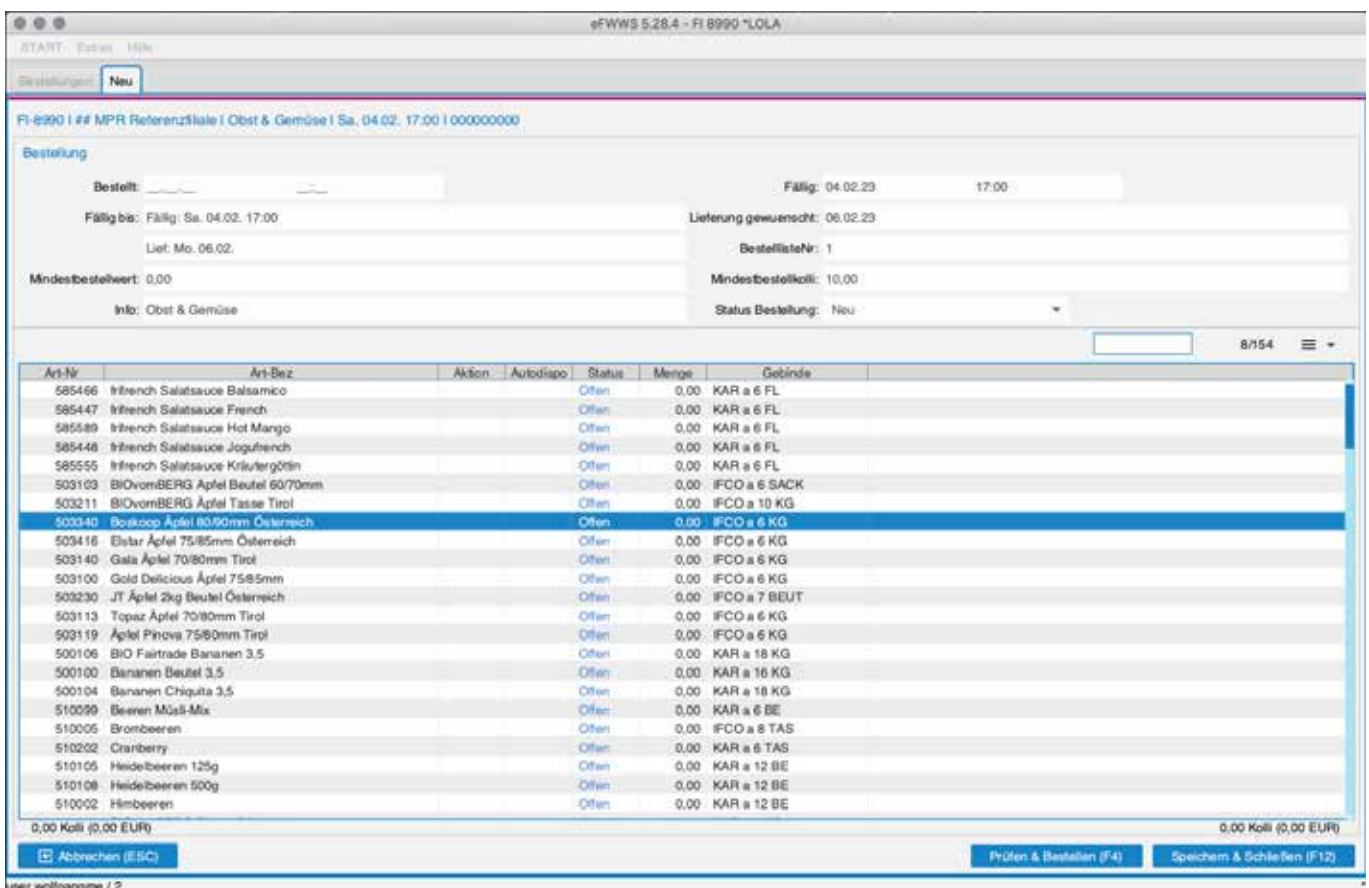


Abb. 4: Generierte Oberfläche

mit minimaler Höhe (-1). In einer zweiten Zeile wird eine Tabelle mit maximaler Höhe gezeigt (1*). Die Tabelle wird mit den Positionen der aktuell „selektierten“ Bestellung geladen. Hinter der Eigenschaft „selektiert“ steht eine Logik, die auch die Abbildung von Master-Detail UIs sehr einfach ermöglicht. Das geht aber an dieser Stelle zu weit. Selbstredend jedoch, dass auch bei dieser DSL im MPS die Code-Completion mit allen, im jeweiligen Kontext möglichen, Vorschlägen zur Verfügung steht. Das erleichtert die Anwendung der DSL enorm, insbesondere für Entwickler mit weniger Erfahrung im Umgang mit dieser DSL. In Abbildung 4 haben wir einen Screenshot der generierten Oberfläche.

Grundsätzlich haben wir auch eine Möglichkeit vorgesehen, um Nicht-Standard-UI-Konzepte in dataUx einzubinden. Interessanterweise sind wir bei den Oberflächen der Warenwirtschaft nicht an Grenzen gestoßen, die derartige Einbindungen erfordern hätten. Alle Anforderungen ließen sich vorerst bestens mit den standardisierten Konzepten umsetzen.

Kommen wir nochmals auf Abbildung 3 zurück. Bei genauer Betrachtung des Tabellen-Konzepts ist ein Verweis auf „Position bearbeiten“ zu erkennen. Dabei handelt es sich um einen Verweis auf ein sogenanntes Kommando, in diesem Fall eben „Position bearbeiten“ aus dem Modell – in Java ein Package – `bestellungBearbeiten`. Das Kommando ist mit der aktuellen Bestellung und mit der, in der Tabelle selektierten Position parametrisiert. Das soll ausdrücken, dass der Endanwender mit einem Klick auf eine spezifische Position in der Tabelle das Kommando „Position bearbeiten“ starten kann. Doch was verbirgt sich hinter dem Konzept des Kommandos?

Kommandos sind nach unserem Verständnis im weitesten Sinne UI-Controller, die die gesamten interaktiven Abläufe mit dem Endanwender steuern. Ein Kommando weist Parameter auf, verfügt über ein Command-Init und hat dann einen oder mehrere Page-Controller. Diese steuern und bedienen die UI-Seiten (PagePanes, s. o.) eines Kommandos. Die Conclusions eines PageControllers stellen die Schaltflächen dar, die am UI des Endanwenders unten rechts sichtbar sind (s. a. Abb. 4). Abbildung 5 ist ein Screenshot des Kommandos aus der MPS IDE. Damit wollen wir nur einen ersten Eindruck der DSL vermitteln. Wir können an dieser Stelle nicht die gesamte Funktionalität erläutern.

Die Erfahrungen haben gezeigt, dass die gute Lesbarkeit der Kommandos das Verständnis der Anwendungsentwickler über die Vorgänge am UI deutlich erhöht. Die Kommandos verbinden die eigentliche Anwendung mit den Domänenmodellen, den dazugehörigen fachlichen Services und der Persistierung (inkl. Transaktionssteuerung). Interessant ist, dass wir grundsätzlich nur vier Typen von Kommandos unterscheiden müssen, um die gesamte Warenwirtschaft zu implementieren. Dabei finden wir in den Kommandos nur noch fachlichen Code. Technische Details und notwendige Laufzeit-Bibliotheken konnten wir vollständig in die Code-Generatoren verschieben. Und gerade im UI-Bereich generieren wir tatsächlich viel Boilerplate-Code.

Die Erfahrungen aus dem Projekt

Domänenspezifische Sprachen sind in der Praxis noch wenig verbreitet. Auch rund um Language Workbenches sind die Communities noch klein. Wir haben vor 12 Jahren mit JetBrains MPS gestartet, eine Laufzeitumgebung, drei DSLs und Code-Generatoren entwickelt, um ein Filial-Warenwirtschaftssystem zu modellieren. Aktuell besteht dieses System aus 451 Kommandos mit 331 UI-Seiten (PagePanes), das heißt, einige Kommandos verfügen über

gar kein UI. Sie führen auf Anweisung des Endanwenders direkt Operationen auf einem der Domänenmodelle aus. Die Domänenmodelle umfassen 118 Entitäten und 85 Value Objects, die von 218 Services und 154 Repositories verwendet werden. Daraus generieren wir 2.657 Java-Dateien mit insgesamt 856.000 Zeilen Produktiv-Code, das heißt exklusive Tests und dem dafür notwendigen Code. Das System läuft auf etwa 550 Android-Geräten mit Barcode-Scanner und knapp 500 Desktop-Computern.

Wir haben die gesamte Warenwirtschaft und deren Komponenten ausschließlich im JetBrains MPS auf Basis von DSLs entwickelt. Zum Einsatz kommen im MPS dabei drei selbst entwickelte DSLs mit jeweils einem Generator, siehe Tabelle 1.

Alle drei DSLs bauen auf der JetBrains MPS Java DSL auf (java baselanguage), von der wir auch das Typensystem und die grundlegende semantische Ausrichtung übernommen haben. Intensiv nutzen wir auch die, an einem funktionalen Ansatz orientierte, Collection-Sprache von JetBrains. Damit formulieren wir fachliche Vorgänge auf Sammlungen. Bis dato sind keinerlei Unzulänglichkeiten aufgefallen. Im Gegenteil: JetBrains MPS ist mit Stand 2022 eine stabile und ausgereifte Language Workbench.



Abb. 5: Kommando „Position bearbeiten“

org.modellwerkstatt.manmap	Bildet die Schnittstelle zur Datenbank, indem Entitäten mit Value Objects auf Tabellen abgebildet werden. Generiert wird aus den Sprachkonzepten SQL-Statements und Objekt-Mapping Code.
org.modellwerkstatt.objectflow	Erlaubt die Modellierung von Domänenmodellen und den fachlichen Services; durch Kommandos können wir sämtliche Abläufe für den Endanwender abbilden.
org.modellwerkstatt.dataUx	Dient der technologie-unabhängigen Modellierung von UIs, inklusive den dazu notwendigen Menüstrukturen, aus denen wiederum Kommandos gestartet werden.

Tabelle 1: DSLs, die auf der JetBrains MPS Java DSL aufbauen

Wie eingangs beschrieben, gab es seitens des Managements Vorurteile gegenüber J2EE-Umgebungen. Wir haben 2010 daher tatsächlich mit einem Low-Tech Java Stack gestartet. Die ersten Anwendungen wurden als JavaFX Rich-Client deployt. Später haben wir unsere Laufzeitumgebung mit Fähigkeiten zur mobilen Datenerfassung erweitert. Smartphones auf Android-Basis mit Barcode-Scanner wurden über serverseitig gerendertes HTML5 integriert. Sowohl die Desktop-Anwendung als auch die MDE-Anwendung wird im MPS mit der dataUx DSL spezifiziert, wenngleich sich die Spezifikationen selbst unterscheiden. PagePanels der MDE-Anwendung sind deutlich schlanker, da der verfügbare Bildschirm-Raum eingeschränkt ist. Mittlerweile haben wir auf Basis des Vaadin-Frameworks auch eine Laufzeitumgebung für einen Web-Client geschaffen, der die JavaFX Rich-Clients ablöst.

Doch wie steht es um die anfangs diskutierten, langfristigen Herausforderungen für das Warenwirtschaftssystem:

- **Architektur:** Unter Einbeziehung von DSLs ist es uns im Projekt Warenwirtschaftssystem für MPREIS gelungen, die technische Architektur vollständig hinter der fachlichen Architektur zu verstecken. Technische Details, die für den Anwendungsentwickler irrelevant sind, konnten wir komplett hinter den DSLs verbergen. Der Anwendungsentwickler kann sich dadurch auf ein „fachliches Java“ in der Domäne konzentrieren. Die Fachlogik der Domäne ist konsequent von der Plattform getrennt, gegen die entwickelt wird. Dabei haben wir einen gewissen Investitionsschutz gegenüber der verwendeten Technologie geschaffen. Es lassen sich Laufzeitumgebung, Datenbank oder UI-Technologie austauschen.
- **Produktivität:** Anwendungsentwickler können stärker auf die fachlichen Probleme fokussieren und bekommen mit MPS und den drei DSLs eine Art Low-Code-Plattform zur Verfügung gestellt, mit der sie sehr effizient und konzentriert an der Warenwirtschaft entwickeln können. Wenn nachvollziehbar ist, wie die DSLs bei der Entwicklung von Anwendungen verwendet werden sollen, wenn die DSLs einfach und intuitiv sind, dann stellt sich auch eine entsprechend hohe Produktivität ein. So wird MPS mit den DSLs mittlerweile auch für andere Anwendungen und zum schnellen Prototypen verwendet. Gerade mit dem Prototyping machen wir jetzt äußerst positive Erfahrungen in anderen Projekten.
- **Entwickler:** Als Anwendungsentwickler können wir auch einen erweiterten Kreis von Personen ansprechen. Das Interesse an der Domäne Warenwirtschaft, an den Domänenmodellen, den fachlichen Operationen und den Abläufen ist deutlich wichtiger und wesentlicher als Wissen um State-of-the-Art Java-Frameworks und UI Toolkits. Dadurch können wir bei der Implementierung Personen involvieren, die über Programmierverständnis verfügen, jetzt aber nicht Hardcore-Java-Ex-

perten für die technische Umsetzung sind. Die DSLs schaffen einen leicht erlernbaren Rahmen in der Anwendungsentwicklung.

Von DSLs und Language Workbenches sollte man sich keinesfalls abschrecken lassen. Sie bieten nach unserer Ansicht ungemeines Potenzial. Auch für die Zukunft halten wir die Verwendung von DSLs in JetBrains MPS für zielführend. Wir können zwei unterschiedliche Lebenszyklen isolieren, wenn wir zwischen Fachlichkeit und Technologie scharf unterscheiden.

Zur Fachlichkeit zählen wir die Domänenmodelle, die Operationen auf diesen Modellen und die vorgestellten Konzepte der Präsentationsschicht. Zur Technologie zählen wir die Laufzeitumgebungen, allen voran aber auch die Code-Generatoren. Unabhängig voneinander können wir beides mit unterschiedlichen Entwicklern über die Zeit hinweg weiter evolutionieren. Die Anwendungsentwickler setzen funktionale Anforderungen in der Domäne mit möglichst hoher Produktivität und Fokus um. Die Entwickler der DSLs und der Laufzeitumgebung schaffen die wiederverwendbare, querschnittliche Infrastruktur für die Anwendungen.

Neue technologische Entwicklungen können wir aufgreifen und in die Plattform integrieren. Architektonische Entscheidungen, die sich nicht bewährt haben, können wir wieder revidieren. Den Anwendungsentwicklern können wir langfristig eine möglichst stabile Plattform bieten, die ihnen die Arbeit in der Domäne erleichtert und den technischen Schulden entgegenwirkt.

Literatur und Links

[Cun92] W. Cunningham, The WyCash Portfolio Management System, Experience Report, OOPSLA 92

[ERP5] <https://www.erp5.com/>

[Eva04] E. Evans, Domain-Driven Design, Addison-Wesley, 2004

[Fow11] M. Fowler, Domain Specific Languages, Addison-Wesley, 2011

[JetB] <https://www.jetbrains.com/mps/concepts/>

[Kho21] V. Khononov, Learning Domain-Driven Design, O'Reilly, 2021

[Lil20] C. Lilienthal, Langlebige Software-Architekturen, dpunkt.verlag, 2020

[Odo0] Open Source ERP and CRM, <https://www.odoo.com/>

[Schä22] A. Schäfer, Domain Specific Languages, in: JavaSPEKTRUM, 2/2022

[Ver13] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2013

[Völ04] M. Völter, Lebendige Architekturen, WhitePaper, 2004

[Völ13] M. Völter, DSL Engineering