

Technische Universität München

Fakultät für Informatik

Diplomarbeit in Informatik

**Überprüfung der Effizienz eines mit
MDA-Werkzeugen generierten
objekt-relationalen Datenbankschema**

Mykhaylo Mukhachov

Aufgabensteller: Prof. Dr. Torsten Grust

Betreuer: Markus Kleinfelder, Loyalty Partner GmbH

Abgabedatum: _____

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum: _____

1	Kundenbindungssystem	4
2	Das Hector System	5
2.1	Architektur des Hector Systems.....	5
2.2	Architektur des Hector Core Subsystems.....	6
2.3	Anforderungen an das Hector System.....	8
3	Model Driven Architecture (MDA).....	10
4	Object-Relational Mapping.....	13
4.1	Hibernate – das Persistence Framework.....	13
4.2	Abbildung von Vererbungen mit Hibernate	15
5	DA-Datenbank.....	17
5.1	Klassenmodell des Hector Core.....	17
5.2	Relationales Modell der DA-Datenbank	19
6	Tests und Analyse der Ergebnisse	24
7	Fazit	28
8	Literatur.....	29

Das Ziel dieser Diplomarbeit besteht darin, die Qualität (im Sinne von Performanz) des von einer Software generierten Datenbankschemas mit der Qualität eines „manuell“ erzeugten zu vergleichen. Dies wurde am Beispiel des Kundenbindungssystems „Hector“ von Loyalty Partner GmbH gemacht.

Folgendes Dokument bietet einen Überblick über das Hector-System und beschreibt die zur Entwicklung des Systems verwendete Technologien wie MDA und Hibernate. Im Weiteren werden unterschiedliche Implementierungsmöglichkeiten der Datenbank, die dem Hector-System zugrunde liegt, betrachtet und getestet. Abschließend werden Testergebnisse analysiert.

Kundenbindungssystem

Als Kundenbindung bezeichnet man die Fähigkeit eines Unternehmens, Kunden zu identifizieren und über Kundenvorteile an das eigene Unternehmen zu binden, um Erlöse und den Profit des Unternehmens zu steigern. Im Bereich des Handels und im Dienstleistungssektor werden den Stammkunden häufig durch Prämien, Bonus oder Rabatte Vorteile gewährt. Als Beispiele von Kundenbindungsprogrammen kann man Vielfliegerprogramme (Miles&More von der Lufthansa, Mileage Plus von United Airlines) oder Kundenkartenprogramme (Payback und HappyDigits in Deutschland, Tesco Clubcard in Großbritannien) nennen. Viele Kundenbindungsprogramme sind auch unternehmensübergreifend: z.B. Meilen von Miles&More kann ein Teilnehmer des Programms nicht nur beim Kauf eines Flugtickets von Lufthansa sammeln, sondern auch beim Erwerb der Flugtickets anderer Fluggesellschaften, die Mitglieder von Star Alliance sind, und sogar bei Hotels, Mietwagenunternehmen usw.

Eine der häufigsten Methoden, einen Kunden zu identifizieren, ist eine Kundenkarte aus Papier oder Kunststoff, die zudem auch einen Magnetstreifen mit hinterlegten Daten enthält. Gegen Vorlage einer solchen Karte erhält der Kunde geldwerte Vorteile, Zusatzleistungen oder andere Vergünstigungen, häufig in Form einer „virtuellen Währung“ wie „Meilen“, „Punkten“, „Digits“. Durch den vermittelten "Spareffekt" kehrt der Kunde zum Unternehmen zurück. Das Unternehmen kann auch das Kaufverhalten des Kunden analysieren und durch die gespeicherten Transaktionsdaten Kundenprofile erstellen.

Das Hector System

Hector ist ein Kundenbindungssystem, das von Loyalty Partner GmbH entwickelt wurde. Das Ziel bei der Entwicklung war es, ein System zu erstellen und auf den Markt zu bringen, das die Daten einer sehr großen Anzahl von Mitgliedern verarbeiten kann. Als Ausgangspunkt dienten die Anforderungen an ein neues Kundenbindungssystem für das „Miles&More“-Programm von Lufthansa. An ein solches System wurden folgende Durchsatzanforderungen gestellt:

- 9 000 000 Mitgliederdaten in der Datenbank;
- 1 000 000 Transaktionen pro Tag;
- 3 000 000 Insert/Update/Delete-Anfragen pro Tag

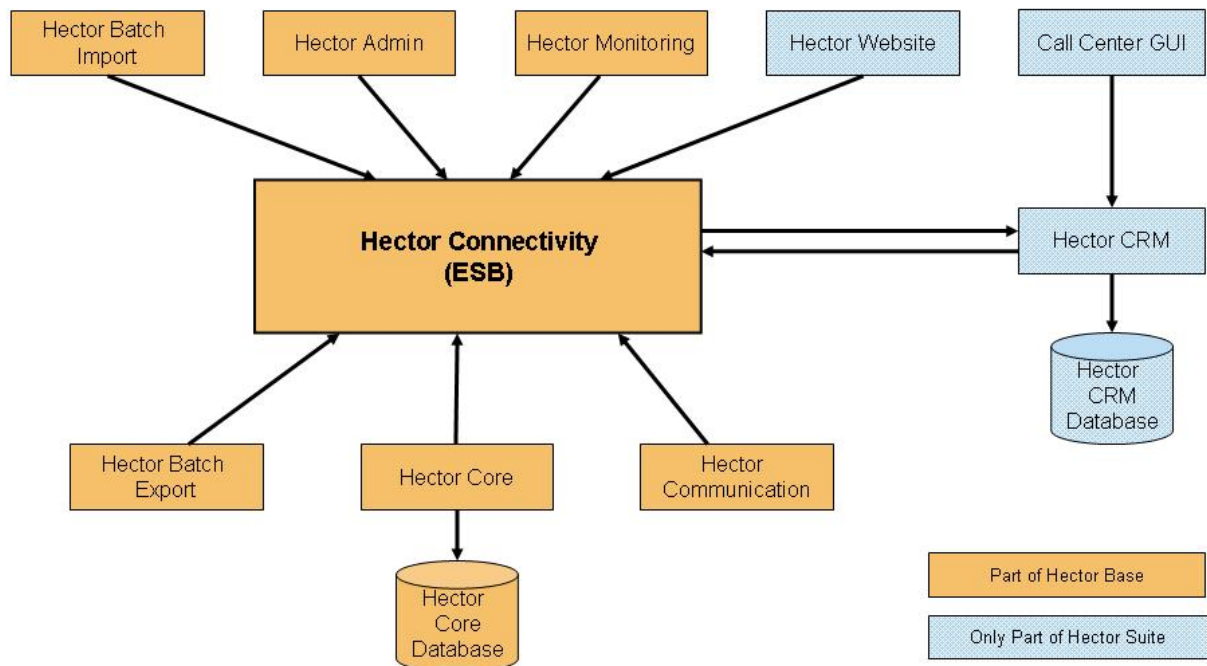


Das Hector System besteht aus drei Teilen: Customer Loyalty Management (CLM) System, Customer Relationship Management (CRM) System und Customer Billing Management (CBM) System. Da die meisten Unternehmen, die an einem Kundenbindungssystem interessiert sein können, bereits ein CRM System besitzen, wurden von Loyalty Partner nur die CLM und CBM Teile entwickelt und das CRM-System des Unternehmens in das Hector System integriert. Außerdem ist es geplant, den Unternehmen, die kein eigenes CRM-System besitzen, eine erweiterte Version des Hector-Systems anzubieten, die „Hector Suite“. Die CLM und CBM Teile des Hector Systems werden nachfolgend als ein Ganzes betrachtet.

Architektur des Hector Systems

Das Hector System enthält zwei für die Business Logik verantwortliche Subsysteme, nämlich Hector Core und CRM System. Das CRM System dient zur Verwaltung der Kundendaten und bietet auch Benutzeroberfläche für den Call Center-Agenten. Wie bereits erwähnt, wird dafür standardmäßig das bereits vorhandene CRM System des Unternehmens verwendet. Das Hector Core System enthält die gesamte Loyalty Business Logik.

Das Hector Connectivity System basiert auf dem ESB (Enterprise Service Bus) Konzept und bietet die Möglichkeit, die Subsysteme (z. B. Hector Core, CRM, Admin...) in das konsistente Hector System zu integrieren. Das Connectivity System implementiert damit Workflows, die sich über mehrere Hector Subsysteme erstrecken.



Das Hector Batch Import System ist für alle Aktivitäten mit Daten-Import verantwortlich. Zum Beispiel, Daten über Collect-Ereignisse, die von Partner-Unternehmen in Form von Dateien täglich übermittelt werden und in das System eingefügt werden sollen.

Analog, das Hector Batch Export System ist für den Export der Daten in der Form von (xml-) Dateien verantwortlich. Daten in dieser Form werden von externen Dienstleistern verwendet, z.B. zur Erstellung von Kundenbriefen.

Das Hector Admin System bietet eine graphische Benutzeroberfläche zur Konfiguration und Verwaltung des Hector Systems. Es ist als Web-basierte Anwendung implementiert.

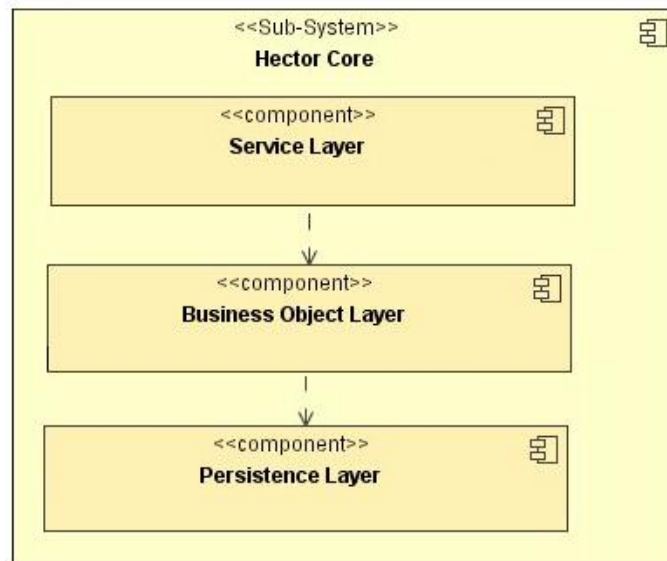
Von allen Teilen des Hector-Systems ist das Hector Core für diese Diplomarbeit von größter Bedeutung.

Architektur des Hector Core Subsystems

Hector Core ist ein Subsystem, das die Billing- und Loyalty-spezifische Business-Logik enthält. Es besteht aus einigen funktionalen Komponenten und Packages, die Business-Objekte beinhalten. Das Core importiert das Persistence Interface und verwendet den DAO (Data Access Objects) Pattern, um auf die Daten in der relationalen Datenbank zuzugreifen. Es exportiert ein Interface, das nach außen einige Services bereitstellt.

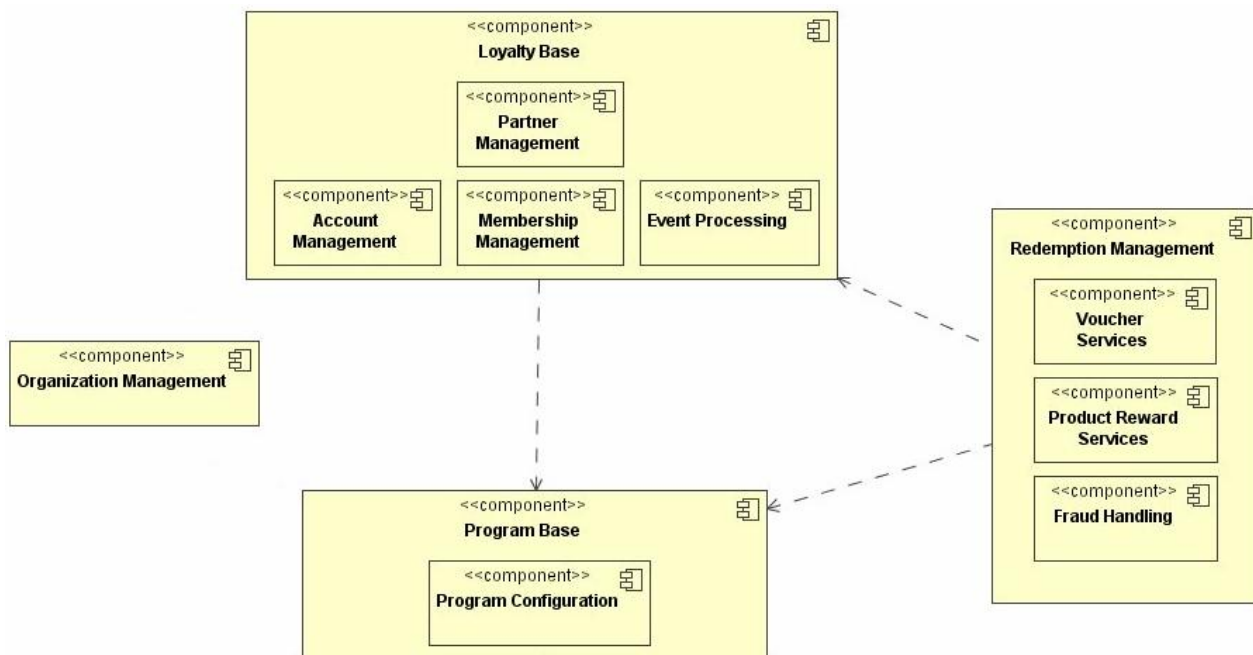
Das Core ist auf einige Schichten (Layers) verteilt:

- Service Layer stellt ein Interface (Facade) zur Business-Logik und Objekte bereit, die von Hector Connectivity benutzt werden;
- Business Object Layer enthält Business-Objekte und ihre Beziehungen zueinander. Auch die grundlegende, nicht veränderbare Business-Logik ist in diesem Layer enthalten;
- Persistence Layer bietet die Möglichkeit, Business-Objekte in einer relationalen Datenbank dauerhaft zu speichern. Dafür enthält der Layer einen objekt-relationalen Mapper (siehe Kapitel 4).



Wie bereits erwähnt, besteht das Hector Core aus mehreren Komponenten, und zwar:

- Loyalty Base Component – enthält die grundlegende Loyalty Business Logik, wie Mitglieder- oder Transaktionsmanagement;
- Program Base Component – enthält Business-Objekte und Logik zur Erzeugung und Verwaltung von Loyalty-Programmen;



- Redemption Management Component – verwaltet die ganze Logik der Einlösung der Units (Einheiten der virtuellen Währungen);
- Organization Management Component – enthält Business-Objekte und Logik zur Erzeugung und Verwaltung komplexer Geschäftsstrukturen.

Die Komponenten „Loyalty Base“ und „Program Base“ sind in einem Package, und zwar LSD (Loyalty Services Domain), zusammengestellt. Die Komponente „Redemption Management“ ist im Package RSD (Redemption Services Domain) enthalten.

Anforderungen an das Hector System

Die ursprünglichen Durchsatzanforderungen an ein neues „Miles&More“-System der Lufthansa sind unten in tabellarischer Form aufgelistet. Da sich diese Diplomarbeit mit Performanz beschäftigt, sind die am häufigsten vorkommenden Use Cases am meisten von Interesse.

Use Case	Beschreibung	Ausführungszeit, s	Volumen pro Tag
<i>Check and Modify Member Status</i>	Überprüfen, ob der derzeitige Status eines Kunden geändert (erhöht, erniedrigt) werden soll und ihn evtl. ändern	<< 1	500000
<i>Authenticate Customer</i>	Ein Mitglied authentifizieren, inkl. Eingabe und Bearbeitung von Kunden- und Mitgliederdaten.	<< 1	100000
<i>Get Account Balance</i>	Aktuellen Kontostand für ein vorhandenes Konto abrufen	<< 1	100000
<i>Get Account Statement</i>	Vollständigen Kontoauszug inkl. Kontostand und detaillierte Transaktionen für ein Konto erstellen (synchrone Abfragen)	< 2	65000
<i>Check Customer</i>	Stammdaten eines Kunden auf Konsistenz, Vollständigkeit und Korrektheit überprüfen	< 1	14000
<i>Check Membership</i>	Mitgliedschaft eines Kunden an einem Programm validieren	< 1	14000
<i>Modify Customer</i>	Stammdaten eines Kunden aktualisieren.	< 1	10000
<i>Create Alias</i>	Einen eindeutigen Identifikator (am häufigsten eine Karte) für einen Kunden/Mitglied erzeugen	<< 1	7000
<i>Create Customer</i>	Stammdaten eines Kunden eingeben	<< 1	6000
<i>Create Membership</i>	Einen Zusammenhang zwischen Kunden und Programm erzeugen	< 1	6000
<i>Create Account</i>	Ein neues Mitgliedskonto erzeugen	<< 1	6000

Die genannten Anforderungen wurden im Laufe von zwei Jahren seit Beginn der Entwicklung des Hector-Systems überarbeitet und angepasst. Die aktuellen Durchsatzanforderungen an das Hector-System sehen folgendermaßen aus:

UseCase	min. Durchsatz, rec/s	max. Latenz ms	min. Durchsatz, rec/h
<i>EarnLoyaltyPoints (Online, vorgerated)</i>	150	100	540000
<i>EarnLoyaltyPoints (Online, ungerated)</i>	100	150	360000
<i>AuthenticateMember</i>	28	50	100800
<i>SetMembershipStatus</i>	10	1000	36000
<i>Get Account Balance</i>	1,2	60	4320

<i>Get Account Statement</i>	0,8	200	2880
<i>Redemption</i>	0,3	500	1080
<i>Enroll Customer (Online)</i>	0,2	330	720
<i>EarnLoyaltyPoints (Batch, ungerated)</i>	100		360000
<i>EarnLoyaltyPoints (Batch, vorgerated)</i>	150		540000
<i>Welcome Communication (Batchexport)</i>	200		720000
<i>Enroll Customer (Batch)</i>	0,2		720

Darüber hinaus sollen **800** Benutzer das System gleichzeitig benutzen können. Auf die einzelnen Use Cases und die Performanzanforderungen an das Hector System wird im Kapitel 6 genauer eingegangen.

Zwei von der Sicht dieser Diplomarbeit aus wichtigsten Ansätze bei der Entwicklung des Hector-Systems sind MDA (Model Driven Architecture) und ORDBMS (Object-Relational DBMS).

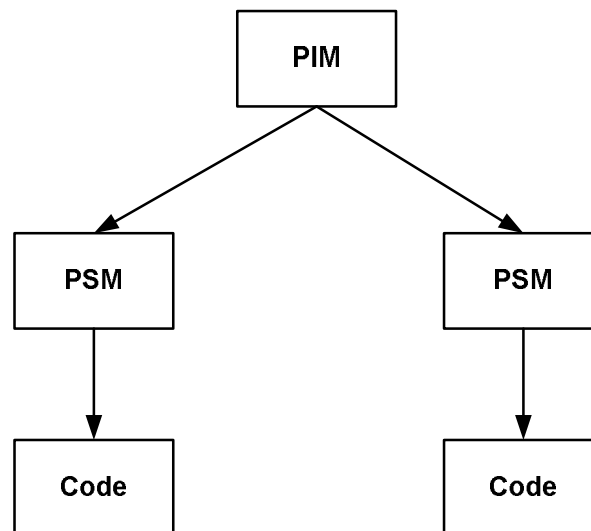
Model Driven Architecture (MDA)

MDA ist ein Ansatz bei der Softwareentwicklung, bei dem der Code ausgehend von einem vorhandenen Modell eines SW-Systems mit Hilfe eines Codegenerators automatisch erzeugt wird. Model Driven Architecture wurde erstmals von Object Management Group (<http://www.omg.org/>) 2001 eingeführt.

Eines der Hauptziele der MDA ist, das Design von der Architektur zu trennen. Da Konzepte und Technologien, die eingesetzt werden, um Design und Architektur zu implementieren, sich unabhängig voneinander entwickelt haben, kann ein Entwickler sowohl für die Architektur als auch für das Design die beste Wahl treffen, wenn diese nicht mehr miteinander gekoppelt sind. Das Design beschäftigt sich mit der funktionalen (Use Cases) Anforderungen, während sich die Architektur mit der Infrastruktur auseinandersetzt, durch die die nicht funktionalen Anforderungen wie Skalierbarkeit, Zuverlässigkeit und Performanz realisiert werden. Der MDA-Ansatz gewährleistet, dass die Plattform Independent Model (PIM), die das konzeptionelle Design des Systems darstellt, eventuelle Änderungen an den Implementierungstechnologien überlebt.

Einen MDA-Prozess kann man in folgende drei Schritte aufteilen:

1. Entwicklung eines plattformunabhängigen Modells (PIM, Platform Independent Model). Das Modell hat ein hohes Abstraktionsniveau und ist an keine Implementierungstechnologie gebunden.
2. Das PIM wird dann in ein oder mehrere plattformspezifische Modelle (PSM, Platform Specific Model) umgewandelt. Das PSM ist an die verwendeten Technologien angepasst, z.B. an ein Datenbanksystem oder EJB.
3. Das PSM wird in den Code transformiert. Da das PSM mit der Implementierungstechnologie fest verbunden ist, ist der Schritt ziemlich trivial.



Die genannten Transformationen werden von Codegenerator-Werkzeugen durchgeführt. Damit wird den Entwicklern die langweilige und fehleranfällige Implementierungsarbeit zum größten Teil abgenommen. Fehler in den generierten Codeanteilen können an einer Stelle – in den Generatorschablonen – beseitigt werden. Außerdem hält die Natur des MDA Ansatzes zu einer sorgfältigeren Konzeption der zu erstellenden Programme in der Entwurfsphase an, was bei sehr komplexer Software von großer Wichtigkeit ist.

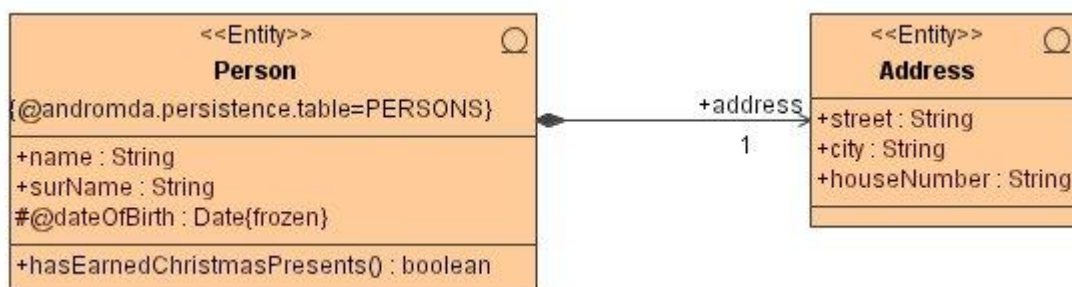
Die Verwendung des MDA Ansatzes stellt damit sicher, dass der Code auf eine konsistente Weise aufgebaut wird, was eine hohe Qualität gewährleistet.

Zur Implementierung der MDA hat sich das Hector Team für das AndroMDA Open Source MDA Framework (<http://andromda.org/>) entschieden. Ein Modell des Hector-Systems (Interfaces, Business-Objekte und ihre Abhängigkeiten voneinander) wird mit der für solche Zwecke „klassischen“ UML unter Verwendung von MagicDraw (<http://magicdraw.com>) beschrieben. Es ist möglich und manchmal auch notwendig, das PIM-Modell mit einigen „Hinweisen“ an den Codegenerator zu versehen, falls ein Element von einem vom Codegenerator vorgeschlagenen Standard abweichen soll. Diese „Hinweise“ werden in der Form von „Tagged Values“ in dem PIM-Modell implementiert. Modellierung des Systems mit dem MagicDraw und UML bietet dem Entwickler eine weitgehende Flexibilität an den Eigenschaften des zu generierenden Datenbankschemas: ein einfaches Beispiel dafür wäre der Fall, wenn die Klasse „Person“ anstatt der vom Codegenerator vorgeschlagenen Tabelle „PERSON“ auf eine in der Datenbank bereits vorhandene Tabelle „PERSONS“ abgebildet („gemappt“) werden soll. In diesem Fall soll der Tabellename explizit in einem Tagged Value angegeben werden. Ein weiterer, oft vorkommender Beispiel ist die Angabe von Attributen (Spalten), die indiziert werden sollen, bereits auf der Modellebene.

Das erzeugte UML-Modell wird dann zur Generation unterschiedlicher Implementierungsartefakten (z.B. Java Klassen, EJB Interfaces, Deployment Deskriptoren usw.) verwendet. Damit wird das Objekt-Modell des Systems im Code abgebildet und nur die Business-Logik des Systems muss "manuell" von den Entwicklern implementiert werden. Das AndroMDA ist mit einigen Codegenerationsmodulen (Cartridges) ausgestattet: Java Cartridge zur Generation von gewöhnlichen Java-Klassen, EJB Cartridge zur Generation von Java Beans, Hibernate Cartridge zur Generation von Persistenzschicht usw. Bei der Bearbeitung des Modells durch AndroMDA werden dessen Elemente auf Übereinstimmung mit bestimmten Stereotypen (z.B. <<Entity>>, <<Enumeration>>) geprüft. Die Cartridges werden dann diese Elemente aufgrund der in der Cartridge Descriptor definierten Templates bearbeiten.

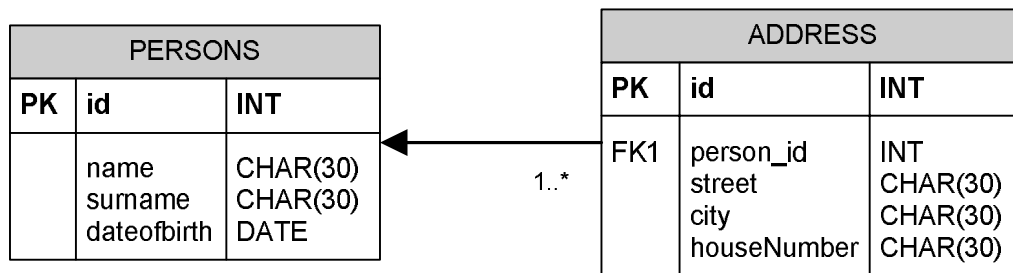
Das AndroMDA Framework ist mit Maven Build Tool (<http://maven.apache.org/>) integriert. Bei einem Aufruf von Maven werden vor dem eigentlichen Build-Vorgang die bestehende UML Modelle untersucht und der Quellcode der Java-Klassen für Objekte des Modells generiert. Außerdem werden Mapping-Dateien für Hibernate Persistenzschicht (siehe Kapitel 4) und ein SQL-Skript zur Erzeugung des Datenbankschemas generiert.

Ein Beispiel zur Generierung eines Datenbankschemas aus dem Klassenmodell mit Maven/AndroMDA/Hibernate: gegeben sei folgendes Klassenmodell



Notiz am Rande: das Modell ist zwar mit der UML beschrieben, aber der Quellcode der Beschreibung ist für einen Menschen kaum lesbar.

Das Model wird dann von AndroMDA (genauer, von dem Hibernate Cartridge) in folgendes Datenbankmodell transformiert:



Die Primär- und Fremdschlüssel Constraints werden zusammen mit den Tabellendefinitionen generiert. Einige Eigenschaften des Schemas muss aber der Entwickler explizit im Modell vorgeben, z.B. dass eine gewisse Spalte Primärschlüssel der Tabelle sein soll, sonst wird vom Cartridge ein Surrogatschlüssel angelegt. Auch die UNIQUE oder CHECK Constraints werden aufgrund expliziter Angabe des Tagged Value in der Beschreibung der Attribute einer Klasse erzeugt. Man sieht an dem Modell einen zugewiesenen Tagged Value für den Tabellennamen: `@andromda.persistence.table=PERSONS`

Selbstverständlich hängt die genaue Form des SQL Codes zur Erzeugung des DB-Schemas von der gewählten RDBMS ab; Das einfachste Beispiel dafür sind Datentypen der Felder. Die Version der SQL-Syntax, die bei der Generation zu verwenden ist, kann in den Einstellungen der Konfigurationsdateien des Hibernate-Cartridges gewählt werden.

Der MDA Ansatz ermöglicht der Entwicklung des Hector-Systems darüber hinaus eine hohe Portabilität des Systems. Gefordert wurde, dass sich das Hector-System mindestens auf der Basis von DBMS Oracle, IBM DB2 und MS SQL Server sowie auf der Basis des Anwendungsservers Bea Weblogic, JBoss und IBM Websphere implementieren lässt. Momentan für die Entwicklung eingesetzte Server sind Oracle 10g und JBoss entsprechend.

Object-Relational Mapping

Der Begriff „Object-Relational Mapping“ bezeichnet die Abbildung von objektorientierten Daten auf relationale Daten und umgekehrt. Damit wird eine „virtuelle objekt-orientierte Datenbank“ erzeugt, die zur dauerhaften Abspeicherung („Persistierung“) von Objekten eines Softwaresystems mit OO-Architektur verwendet werden kann.

Das Bedürfnis an einem Verfahren, mit dem sich die OO-Welt auf die RDBMS-Welt abbilden lässt, ist durch grundlegenden Unterschiede zwischen den Paradigmen objekt-orientierter Software und relationalen Datenbanken bedingt, wie z.B. der Unterschied zwischen der Collection in OO-Software und der Tabelle in einer RDBMS. Ein anderes Beispiel wäre eine many-to-many Relation, die zwischen zwei Objekten ganz natürlich ist, in einer RDBMS aber mit einer extra Tabelle implementiert werden muss.

Damit sich ein Entwickler von OO-Softwaresystem nicht um Persistierung jedes Objektes bei jeder Datenoperation (Abspeicherung einer neuen Instanz eines Objektes, Aktualisierung einer bereits bestehenden Instanz usw.) kümmern muss, werden Objekte des Systems auf die Datenbankstrukturen einmal fest abgebildet (gemappt) und die Datenpersistierung wird einem SW-Subsystem überlassen. Damit sieht ein Entwickler die *relationale* Datenbank nur als Speicher für *Objekte*. Wie beim MDA-Ansatz wird damit die Qualität des Codes erhöht und gleichzeitig der Zeit- und Arbeitsaufwand bei der Entwicklung verringert.

In einer schichtweise organisierten Architektur ist für das Mapping üblicherweise eine so genannte Persistenzschicht (Persistence Layer) verantwortlich.

Hibernate – das Persistence Framework

Für die Entwicklung des Hector-Systems wurde das Hibernate (<http://hibernate.org>) – Open Source Object-Relational Persistence Framework – verwendet. Das Framework ist Java-basiert, aber auch eine .NET-Portierung existiert. Das Hibernate wandelt die Aufrufe von Java-Klassen und Methoden (typisches Beispiel aus dem Code von Hector:

`getMembership().getCollectsTo().getBalanceDetails(virtualCurrency).getBalance ()`) in SQL-Abfragen an die Datenbank um. Im obigen Beispiel wird eine Reihe von SELECT Abfragen erstellt, DML-Abfragen werden selbstverständlich auch vom Hibernate generiert. Neben einfacher Abbildung von Klassenoperationen auf die CRUD-Abfragen stellt das Hibernate eigene Abfragensprache HQL zur Verfügung, die es einem Entwickler ermöglicht, komplexe Anfragen an die Datenbank zu erstellen. Dabei ist zu beachten, dass trotz einer (absichtlichen) Ähnlichkeit zwischen HQL und SQL, die HQL objekt-orientiert ist, d.h. Vererbung, Polymorphismus und Assoziationen werden von der HQL völlig unterstützt.

Die Abbildung einer Klasse auf die Datenbankstrukturen wird in einer so genannten Mapping-Datei angegeben. Mapping-Dateien sind in einem für einen Menschen lesbaren XML Format erstellt. Im Wesentlichen enthält eine Mapping-Datei folgendes:

- die Tabelle, auf die die Klasse abgebildet werden soll:

```
<class name="com.loyaltypartner.hector.bol.membership.AliasImpl"
table="MM_ALIAS" discriminator-value="Alias" abstract="false">
```

- die Angabe der Primärschlüssel:

```
<id name="id" type="long" unsaved-value="null">
  <column name="ALIAS_ID"/>
```

```

    <generator class="seqhilo">
      <param name="sequence">SEQ_MM_ALIAS</param>
      <param name="max_lo">100</param>
      <param name="parameters">START WITH 1000</param>
    </generator>
  </id>

```

- die Zuweisungen der Spalten an die Klassenattribute:

```

<property name="pin" type="string" column="PIN" not-null="true"
length="4"/>

```

- die Angabe der Fremdschlüsselabhängigkeiten:

```

<many-to-one name="program"
class="com.loyaltypartner.hector.bol.program.ProgramImpl"
index="ALIAS_IDPRGTYPE_UK" lazy="proxy" fetch="select">
  <column name="PROGRAM_ID" not-null="true"/>
</many-to-one>

```

Die Eigenschaften von Hibernate Framework, die für diese Diplomarbeit wegen ihres Performanceinflusses vom größten Interesse sind, sind Nebenläufigkeitssteuerung („Concurrency control“) und Datenabrufstrategien („Fetching strategies“).

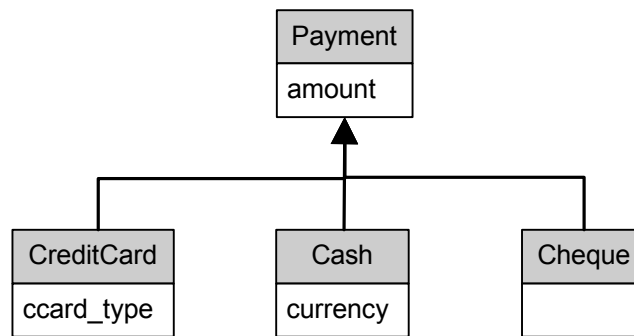
Die Transaktionsisolationsebene wird im Hibernate über globale Konfigurationseinstellung (`hibernate.connection.isolation`) festgelegt. Für die Applikationen mit hohen Nebenläufigkeitsanforderungen bietet das Hibernate die so genannte „Optimistic concurrency control (OCC) with versioning“ Technik. Bei der globalen Einstellung von „Read Committed“-Isolationsebene bietet der Einsatz dieser Technik (kombiniert mit der obligatorischen Verwendung von Context Cache) viele Vorteile von „Repeatable Read“ an, einer der wichtigsten davon ist das gelöste „Lost Update“-Problem. Grundsätzlich sieht (von der Sicht eines DB-Entwicklers) die „OCC with versioning“ folgendermaßen aus: die Datenbankentitäten werden mit einer Versionsspalte (Versionsnummer oder Timestamp) ausgestattet. Wird die Instanz eines Objekts aktualisiert, erhöht das Hibernate die Versionsnummer. Bei nebenläufigen Aktualisierungen werden die Versionen vom Hibernate automatisch miteinander verglichen. Kommt es zu einem Konflikt, wird die spätere Aktualisierung rollbacked und eine Ausnahme (Exception) geworfen. Damit wird das „Lost Update“ verhindert.

Eine „Fetching Strategie“ ist eine Strategie, die vom Hibernate zum Abrufen von mit einem bestimmten Objekt assoziierten Objekten eingesetzt wird. Die Attribute „fetch“ und „lazy“ (siehe Beispiel oben) spezifizieren, *wie* die entsprechende Kollektion abgefragt wird (SQL-mäßig: OUTER JOIN in gleichen SELECT wie die Instanz selbst; eine zweite SELECT-Abfrage nur für diese Instanz; eine zweite SELECT für alle Instanzen aus der vorherigen Abfrage; usw.) und *wann* die Abfrage stattfindet (die ganze Kollektion gleichzeitig mit der Instanz des Objektes; die ganze Kollektion, aber erst wenn die Applikation eine Operation auf der Kollektion auszuführen versucht – dies ist eine Default-Einstellung; einzelne Elemente der Kollektion, wenn die Operationen auf diesen Elemente gestartet werden; usw.). Die Fetching Strategie ist für jede Assoziation einzeln zu definieren und wird in einer Mapping-Datei angegeben. Es ist auch möglich, die in der Mapping-Datei festgelegte Strategie beim Ausführen einer HQL-Abfrage aufzuheben. Die richtige Auswahl der Fetching Strategie kann den Datenfluss zwischen der Datenbank und der Applikation deutlich reduzieren und damit die Performanz erhöhen.

Abbildung von Vererbungen mit Hibernate

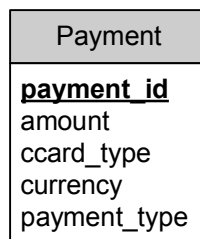
Die am einfachsten zu merkende Differenz zwischen der objektorientierten Welt und der Welt der relationalen Datenbanken ist, dass zwischen den Objekten der OO-Welt sowohl „has a“ als auch „is a“ Relationen, während zwischen den Objekten der relationalen Welt nur „has a“ Relationen bestehen.

Die Implementierung von Generalisierungen/Vererbungen in einer mit Hilfe von Hibernate O/R-gemappten DB ist auf mehrere (genau 4) Arten möglich. Betrachten wir als Beispiel folgendes Klassenmodell - eine abstrakte Klasse „Payment“ mit drei Erben:



Dieses Modell kann auf folgende Weisen auf eine relationale Datenbank abgebildet werden:

- „Table per class hierarchy“:
die abstrakte Klasse ist zusammen mit allen konkreten Klassen in einer Tabelle dargestellt, die als Spalten Attribute aller konkreten Klassen hat. Das Selektieren der richtigen konkreten Klassen ist mit der expliziten Angabe der discriminator-Spalten (in dem Beispiel – „payment_type“) in der Mapping-Datei von Hibernate gewährleistet. Standardmäßig entspricht der Wert dieser Spalte dem Namen entsprechender konkreter Klasse.



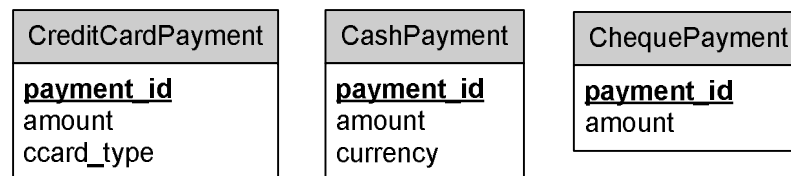
. Laut Literatur (1, s.99; 2, s.199), ist die Strategie ein Gewinner im Sinne von Performanz. Diese Strategie hat auch Nachteile: z.B. die nicht übereinstimmende Attribute der konkreten Klassen müssen als nullierbare Spalten deklariert werden, selbst wenn sie nicht nullierbar (NOT NULL) sein sollen. Zu beachten: Mit dieser Strategie ist das Datenbankmodell denormalisiert!

- „Table per subclass“:
Mit dieser Strategie werden die „is a“-Relationen (d.h. Vererbungen) durch den „has a“-Relationen (Fremdschlüssel) dargestellt. Jede Klasse ist auf eine eigene Tabelle abgebildet, die Vererbungsbeziehung ist mit einer 1:1 Primärschlüssel-Fremdschlüssel-Relation implementiert. Die Attribute jeder Klasse sind als Spalten entsprechender Tabelle gemappt. Die Instanzen einer konkreten Klasse können dann mit einem JOIN von Tabellen von Unter- und Oberklassen geholt werden. Die Aufgabe, die JOIN-Abfrage zu erstellen, ist der Hibernate-Schicht zu überlassen.

Wie bei der vorherigen Strategie, ist es hier ebenfalls möglich, eine discriminator-Spalte in der Tabelle anzugeben, auf die die abstrakte Klasse gemappt ist. Im Unterschied zu einigen anderen O/R-Mapping Software erfordert es das Hibernate Framework nicht, aber z.B. wegen der Kompatibilität mit bestehenden Datenbankstrukturen ist diese Möglichkeit wertvoll.

Der Hauptvorteil dieser Strategie besteht darin, dass sie ein normalisiertes Datenbankschema anbietet. Andererseits wird bei der mehrschichtigen Klassenerbstrukturen eine Abfrage an eine konkrete Klasse aus den tieferen Schichten der Struktur zur mehreren JOINS über mehrere Tabellen führen, was die Performanz der Datenbank senken kann.

- *“Table per concrete class (with unions)”*: jede konkrete Klasse ist als eine Tabelle dargestellt, die von den abstrakten Klassen vererbten Attribute kommen in jeder Tabelle vor, die abstrakte Klasse ist als UNION zu betrachten. Die Aufgabe, eine solche UNION zu erstellen, ist dem Hibernate zu überlassen.



Ein Problem, das sich bei dieser Art von Mapping ergibt, besteht darin, dass polymorphe Assoziationen schlecht unterstützt werden. Da solche Assoziationen normalerweise als eine Fremdschlüsselbeziehung auf der Datenbankebene implementiert sind, muss im Fall einer Assoziation mit der abstrakten Klasse eine Beziehung zu jeder „concrete class“-Tabelle erzeugt werden.

Auch polymorphe Abfragen (an die abstrakte Klasse) sind schwer zu implementieren: Es müssen mehrere SELECT's durchgeführt werden, jeweils zur jeder „concrete class“-Tabelle. Um Performanz zu verbessern, könnte man diese SELECT-Abfragen mit UNION implementieren, für Schreibzugriffe muss Hibernate jedoch mehrere SQL-Anweisungen verwenden (die UNION's werden nicht unterstützt).

- *“Table per concrete class (with implicit polymorphism)”*: der Fall unterscheidet sich von dem Fall *“Table per concrete class (with unions)”* nur in dem Inhalt von Mapping-Dateien, die Datenbankebene wird auf die gleiche Weise implementiert. Der Nachteil dieser Strategie liegt darin, dass Hibernate bei der Ausführung von polymorphen Abfragen keine UNIONS generiert.

Grundsätzlich lässt das Hibernate eine Mischung der Strategien nicht zu: Eine für Implementierung einer Klassenhierarchie gewählte Strategie muss für alle Klassen der Hierarchie verwendet werden. Es existiert jedoch eine Ausnahme von der Regel: man darf die „Table per hierarchie“ und „Table per subclass“ mischen. Man kann die ganze Hierarchie auf eine Tabelle abbilden und eine Subklasse auf eine weitere Tabelle „auslagern“.

Die Auswahl zwischen den aufgelisteten Strategien ist bei der Generation des Codes durch Änderung der Einstellung *„hibernateInheritanceStrategy“* in dem Hibernate-Cartridge von AndroMDA möglich.

DA-Datenbank

Laut Aufgabenstellung sollte in dieser Diplomarbeit die Effizienz des von MDA-Werkzeugen generierten Datenbankschemas ausgewertet werden. Dafür wurde Folgendes gemacht:

- das Objektmodell von *Hector Core* analysiert und anhand der Analyse ein relationales Datenbankschema („DA-Datenbank“) erstellt;
- ein Satz von Mapping-Dateien zum objekt-relationalen Mapping des bestehenden Hector Core Subsystems auf die erzeugte relationale Datenbank erstellt. Damit wurde das Hector System an die DA-Datenbank gebunden;
- ein Set der für das *Hector System* vorgesehenen Durchsatztests auf einer im vorherigen Schritt aufgebauten Variante des Hector Systems durchgeführt;
- Ergebnisse der Tests analysiert.

Die Hector Core Datenbank enthält aufgrund der Natur des objekt-relationalen Mapping keine aktiven DB-Strukturen wie Trigger oder gespeicherte Prozeduren, sondern nur Datenobjekte wie Tabellen, Indizes und Constraints. Wie bereits erwähnt, ist die gesamte Business-Logik im Hector Core Subsystem (im Business Object Layer) implementiert.

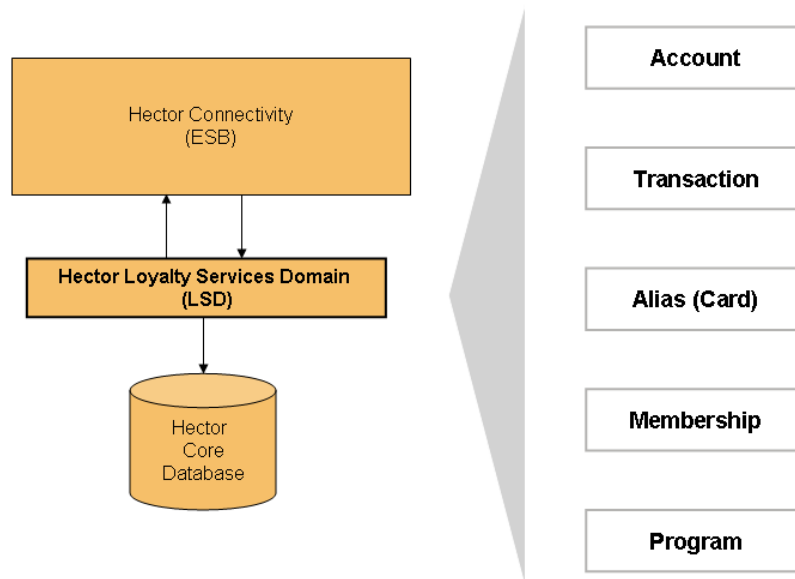
Die Aufgabe, eine neu entwickelte Datenbank an ein bestehendes System anzuschließen, legt einige Beschränkungen bei der Implementierung der Datenbank fest. Zum Beispiel ist es im Prinzip möglich, benutzerdefinierte Typen in einer Java-Applikation an die benutzerdefinierte Datentypen (UDT, user-defined types) in einer Datenbank mit der Hibernate Mapping-Dateien abzubilden dafür ist jedoch pro Typ eine zusätzliche Klasse in der Anwendung zu erstellen. Leider ist es nach Aufgabenstellung nicht möglich, den Code des Hector Systems zu ändern. Ein weiteres Beispiel für Beschränkungen dieser Art wurde bereits erwähnt: das Hibernate lässt es nicht zu, unterschiedliche Strategien zur Vererbungsimplementierung innerhalb einer Klassenhierarchie zu verwenden. Die beiden Möglichkeiten wurden im Laufe der Entwicklung der DA-Datenbank betrachtet und aus den oben genannten Gründen verworfen.

Klassenmodell des Hector Core

Nach Analyse des bestehenden Klassenmodells von Hector Core wurde entschieden, die größte (im Hinblick auf die direkten Abhängigkeiten und Vererbungen) zusammenhängende Komponente von Core, nämlich Loyalty Services Domain (LSD), zu implementieren. Da die meisten Use Cases mit hohen Durchsatzanforderungen die Objekte dieser Komponente betreffen, ist sie im Hinblick auf Performanzmessungen von besonderer Wichtigkeit.

Eine vollständige Beschreibung der Entitäten von LSD-Komponente und ihrer Relationen zueinander ist im Anhang zu finden, wobei hier nur die wichtigsten Klassen (Entitäten) beschrieben werden:

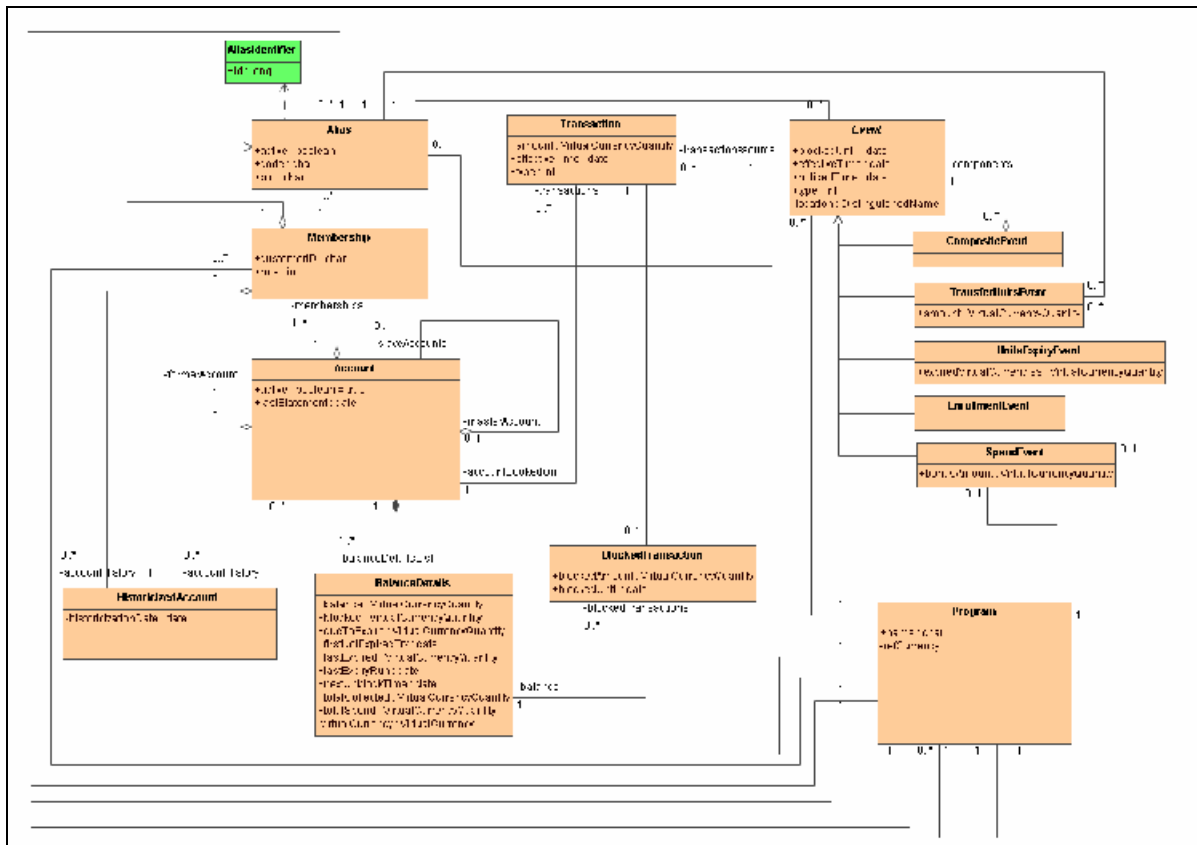
- *Membership* – Darstellung einer Person aus der realen Welt in einem System. Stellt einen Zusammenhang zwischen der Person und dem Programm her;
- *Program* – eine Hauptentität des Hector Systems. Stellt das Kundenbindungsprogramm dar;
- *Account* – das Mitgliedskonto im System beinhaltet alle Transaktionen. Das Mitglied sammelt Units („Punkten“, „Meilen“, usw.) auf das Konto und gibt Units von seinem Konto aus;
- *Alias* – ein Mittel zur Identifizierung eines Mitglieds im System. Repräsentiert eine „physische“ Identifizierungsmethode (z.B. Karte);
- *Transaction* – jede Änderung an der Kontobilanz eines Mitglieds wird in der Datenbank durch eine Transaktion dargestellt.



Laut Anforderungen an das Hector System sind Anfragen an die Haupttabellen der LSD-Komponente folgendermaßen verteilt:

Table / entity	Nr of rows	Insert, %	Update, %	Delete, %	Select, %	Query / DMLfacto
Account	15000000	1	44	0	44	1,222
Alias	30000000	1	1	0	98	49,0
Balance Details	30000000	2	49	0	49	0,96
Event	700000000	25	0	0	75	3,0
Historized membership status	30000000	1	0	0	99	99,0
Membership	30000000	1	0	0	99	99,0
Transaction	900000000	25	0	0	75	3,0

Folgender Ausschnitt aus dem Klassendiagramm zeigt die wichtigsten Entitäten des Systems und ihre Relationen zueinander. Das vollständige Diagramm befindet sich im Anhang.



Das sind Entitäten, die an den meisten Abfragen der Datenbank teilnehmen. Daher ist ihre Implementierung für die Datenbankperformanz von größter Bedeutung.

Relationales Modell der DA-Datenbank

Die meisten Klassen der LSD-Komponente von Hector Core lassen sich nur auf eine einzige Weise sinnvoll auf eine Datenbankebene abbilden, unabhängig davon, ob dies manuell oder von einem Codegenerator gemacht wird. Daher hat ein manuell erzeugtes relationales Modell der Datenbank relativ große Ähnlichkeit mit dem generierten Modell: Die Abbildung erfolgt nach dem Prinzip „eine Klasse – eine Tabelle“. Unwesentliche Abweichungen von diesem Prinzip sind bei der Abbildung von Klassen festzustellen, die den Stereotyp <<Dynamic>> implementieren (z.B. „Membership“): Die dynamischen Attribute einer Klasse werden in einer Sondertabelle gespeichert, die eine Fremdschlüsselbeziehung zur Haupttabelle der Klasse hat. Aber auch diese Klassen werden sinngemäß sowohl in einem manuell als auch in einem automatisch erzeugten Datenbankschema auf die gleiche Weise abgebildet.

Im Laufe der Analyse wurde beschlossen, einige Indexe auf den Fremdschlüsselspalten zu erstellen, die in der generierten Hector Core Datenbank ursprünglich nicht vorhanden waren. Insbesondere sollten die Spalten MEMBERSHIP.ACCOUNT_ID und BALANCE_DETAILS.ACCOUNT_ID (siehe die Modelle weiter) indiziert werden. Da das Hector System sich immer noch in der Entwicklung befindet, wurden die Indexe während der Testphase auch in die Hector Core Datenbank (mit der Angabe von Tagged Values in dem Modell) übernommen. Auf die Erstellung weiterer möglicher Indexe auf den Fremdschlüsselspalten wurde nach der Codeanalyse verzichtet: Es wurden keine Methoden/Abfragen gefunden, bei deren Ausführung ein solcher Index Anwendung finden könnte. Ständige Aktualisierung eines unnötigen Indexes könnte dabei die Performanz nur senken.

Da das Hybernate Framework unterschiedliche Strategien zur Implementierung der Vererbungen anbietet, wurde beschlossen, zwei Varianten der DA-Datenbank zu erzeugen, und zwar eine Variante unter Einsatz der „Table per hierarchy“-Strategie und die andere unter Einsatz der „Table per subclass“-Strategie. Wegen Kompatibilität mit dem vorhandenen Code der Anwendung sollte die zweite Variante in Form „Table per subclass (with discriminator)“ ausgeführt werden. Unterschiede zwischen den beiden Implementierungsarten werden an folgenden Ausschnitten aus den relationalen Schemas dargestellt. Jede der aufgelisteten Klassenhierarchien weist Besonderheiten auf, die unterschiedliche Aspekte des O/R-Mapping repräsentieren.

Die Klassenhierarchie „Alias“ (siehe das Klassendiagramm) stellt einen typischen Beispiel für Unterschiede zwischen den beiden gewählten Strategien dar: In der ersten Variante werden alle Subklassen zusammen mit der Oberklasse auf eine Tabelle abgebildet, in der zweiten wird jede Klasse der Hierarchie auf eine eigene Tabelle abgebildet.

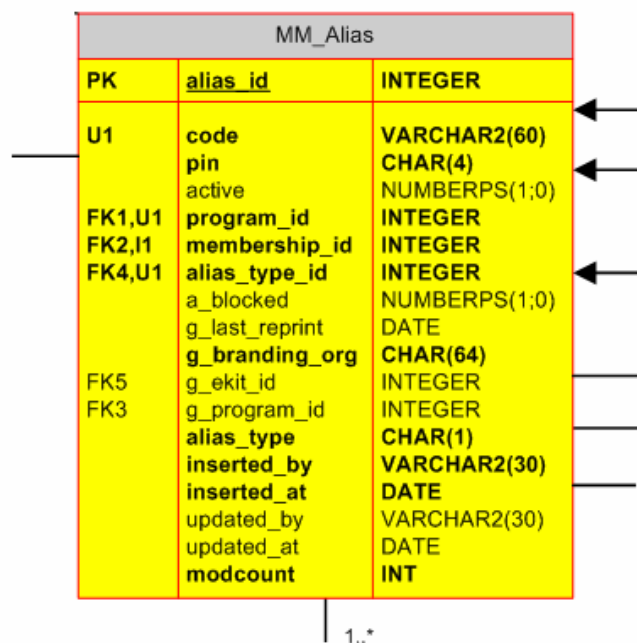


Table per hierarchy: Alias

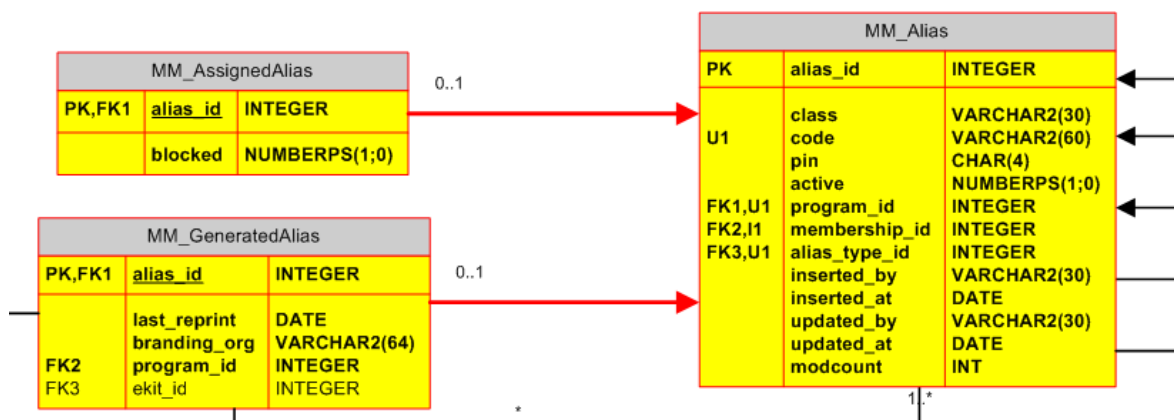


Table per subclass: Alias

Die Besonderheit der Klassenhierarchie „Event“ liegt darin, dass sie das „Composite Pattern“ implementiert. Auf der Datenbankebene im Fall von „Table per hierarchy“ wird die Hierarchie mit einer Tabelle dargestellt, die mit sich selbst eine Fremdschlüsselrelation hat.

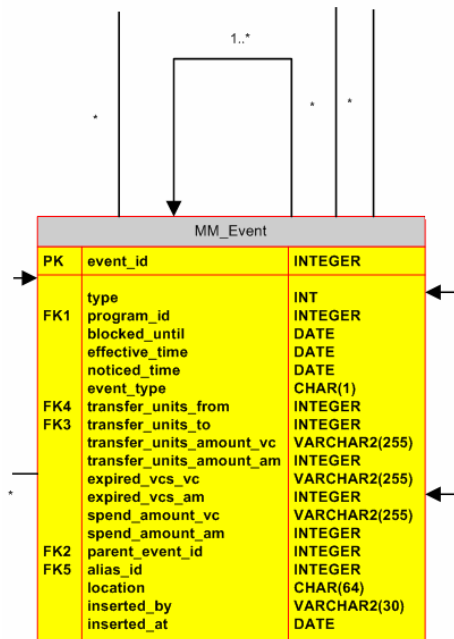


Table per hierarchy: Event

Im Fall von „Table per subclass“ ist die „Composite Event“-Klasse mit einer separaten Tabelle dargestellt, die mit der Haupttabelle von der „Event“-Klasse in gegenseitigen Fremdschlüsselbeziehungen steht.

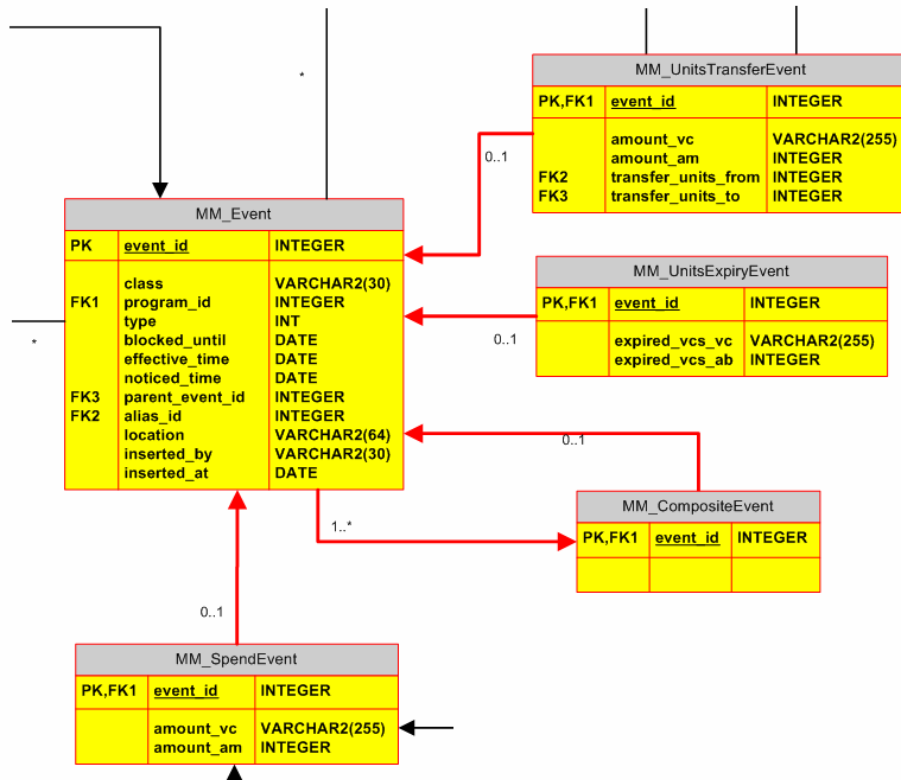


Table per subclass: Event

Die Klassenhierarchie „LookupValue“ weist einige Besonderheiten auf: Zum Ersten hat sie eine mehrstufige Vererbung („LookupValue“->„AliasType“->„GeneratedAliasType“, „AssignedAliasType“) und die Klasse in der Mitte dieses Hierarchieastes („AliasType“) besitzt keine eigene Attribute. Daher ist dieser Klasse keine Tabelle in der Datenbank zugewiesen. Die zugehörige Mapping-Datei dient nur dazu, Abhängigkeiten zwischen den Tabellen festzulegen, und gibt kein echtes Mapping an.

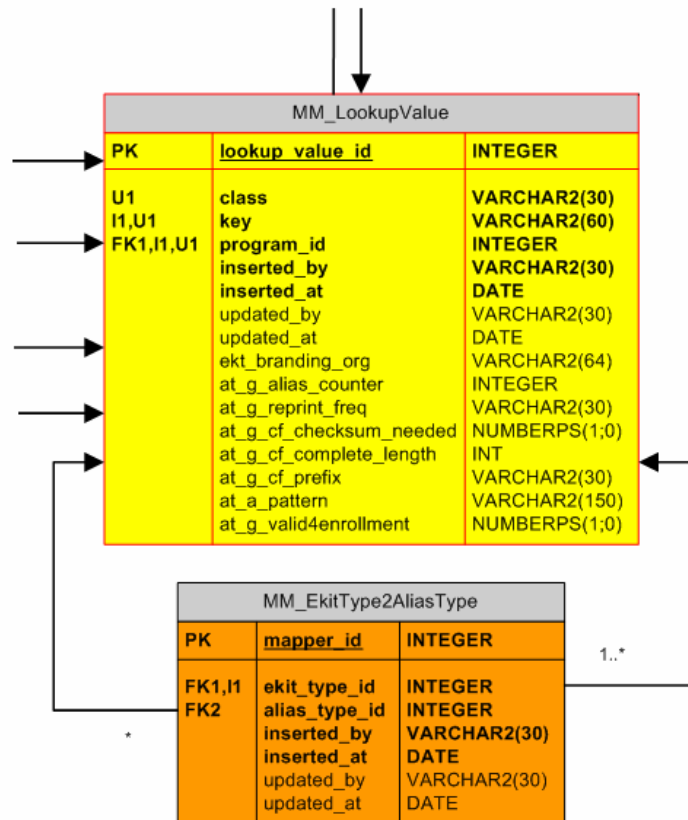


Table per hierarchy: LookupValue

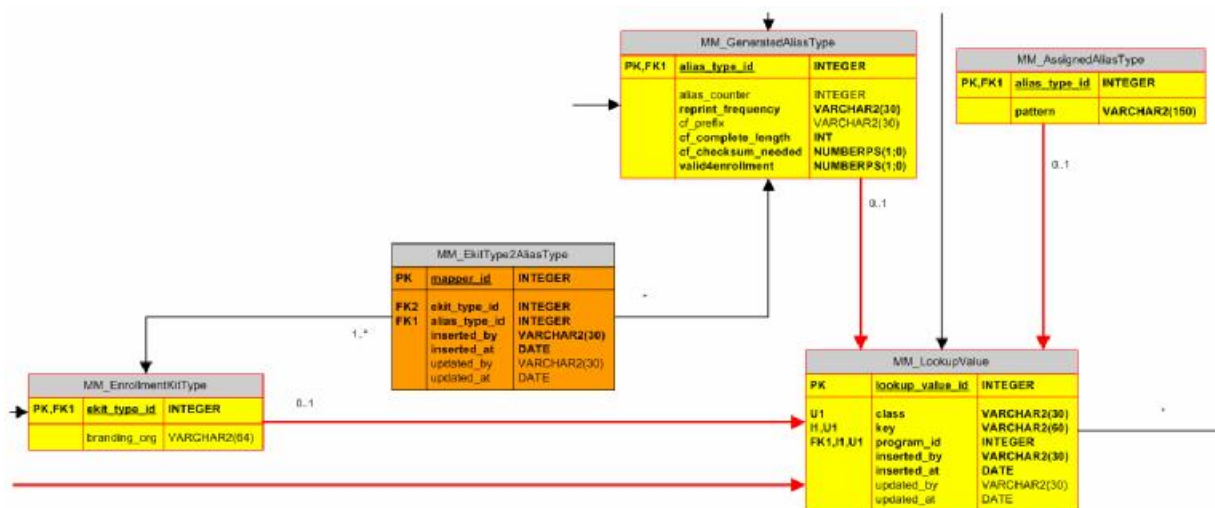


Table per subclass: LookupValue

Zum Zweiten besteht zwischen den Unterklassen dieser Hierarchie eine „many-to-many“ Relation, die mit einer Assoziationstabelle implementiert wird.

Zur Verbesserung der Performanz des erzeugten Schemas wurden verschiedene Möglichkeiten in Betracht gezogen:

- für die Tabellen, die in einer Hierarchie eine Oberklasse darstellen, wurde die Möglichkeit betrachtet, einen Bitmap-Index auf der Discriminator-Spalte zu erstellen. Diese Lösung wurde aber verworfen, da nach der Analyse des Codes von Hector Core klar geworden ist, dass es keine Methoden/Anfragen gibt, die einen Index wie diesen beim Aufruf verwenden können;
- für die gleichen Tabellen wurde die Möglichkeit der Partitionierung („Partitioning pruning“) betrachtet. Die Partitioning wird von dem DBMS zusammen mit jeder Index-, Join- oder paralleler Zugriffstechnik eingesetzt. Demzufolge wurde die so genannte „List partitioning“ für die Tabellen „Alias“ und „Event“ in der DB-Variante „Table per hierarchy“ eingesetzt. Mit dieser Technik wird eine Tabelle physisch auf mehrere Speichereinheiten aufgeteilt und die Einträge werden in unterschiedlichen Einheiten gespeichert, abhängig von dem Wert in der so genannten „partitioning-key“-Spalte. In dem DB-Schema wurden jeweils discriminator-Spalten als Partitioning key gewählt. Damit wurde sichergestellt, dass Instanzen unterschiedlicher Subklassen einer Oberklasse in den unterschiedlichen Speichereinheiten gespeichert werden;
- das Hibernate kann ein so genanntes ROWID verwenden für den schnellen Update-Zugriff auf die Tabellen in einem DBMS, die dieses Feature unterstützen. Das ROWID charakterisiert die physische Lage des entsprechenden Eintrags auf einem Speichermedium. Um diese Möglichkeit auszunutzen, muss man eine entsprechende Angabe in einer zu der Tabelle gehörenden Mapping-Datei machen. In der DA-Datenbank wurde diese Möglichkeit für zwei Klassen implementiert, bei denen auf die UPDATE-Anfragen ein großer Anteil aller Anfragen entfällt, und zwar für „Account“ und „BalanceDetails“ (siehe Tabelle in Kap. 5.1).

Die letzte Möglichkeit zur Performanzverbesserung gehört natürlich nicht zu den Eigenschaften einer relationalen Datenbank, sondern zu den Eigenschaften der Persistenzschicht. Weitere Möglichkeiten zur Verbesserung des Durchsatzes der Persistenzschicht (wie z.B. mögliche Änderungen an den Fetching Strategien) im Vergleich zum ursprünglichen Persistenzschicht-Durchsatz des Hector Systems wurden nicht betrachtet, da laut Aufgabenstellung die Qualität des generierten DB-Schemas und nicht die der Persistenzschicht ausgewertet werden soll.

Wie bereits erwähnt, haben beide im Laufe der Diplomarbeit erzeugten Datenbankschemas und das ursprüngliche DB-Schema des Hector Core Subsystems eine erkennbare Ähnlichkeit. Daher wurden auch relativ kleine Änderungen an den Mapping-Dateien des ursprünglichen Hector Systems vorgenommen, um das Hector Core an neu erzeugte Datenbankschemas anzuschließen.

Tests und Analyse der Ergebnisse

Die Use Cases (UC), an die die höchsten Durchsatzanforderungen gestellt werden, sind folgende:

- EarnLoyaltyPoints
- SetMembershipStatus
- EnrollCustomer
- Authenticate
- GetAccountBalanceDetails
- GetAccountStatement
- Redemption

Bei UC EarnLoyaltyPoints werden für das angegebene Mitglied (Member) Events und Transaktionen (PurchaseEvents, PurchaseItemEvents und CouponEvents) im angegebenen Programm gespeichert.

Bei UC SetMembershipStatus wird für das angegebene Mitglied der entsprechende Status im angegebenen Programm gesetzt.

Bei UC EnrollCustomer wird ein Customer in Hector neu angelegt (UC CreateCustomer). Danach wird für einen bestehenden Customer eine neue Mitgliedschaft für ein gegebenes Programm in Hector angelegt (UC CreateMembership).

Bei UC Authenticate wird für das angegebene Mitglied eine Authentifizierung am Hector-System durchgeführt.

Bei UC GetAccountBalanceDetails wird die aktuelle Bilanz des jeweiligen Accounts im angegebenen Programm zurückgegeben.

Bei UC GetAccountStatement wird eine Liste von Transaktionen (alle Transaktionen seit dem letzten Aufruf von GetAccountStatement) des jeweiligen Accounts im angegebenen Programm zurückgegeben.

Bei UC Redemption werden für das angegebene Mitglied die Einlösungsevents und Transaktionen (SpendEvents, ProductRewardEvents, VoucherEvents) im angegebenen Programm gespeichert.

Die Tests werden auf einer 3-schichtigen Architektur durchgeführt:

- Der Testclient kann auf beliebigen Test- oder Entwicklungsrechnern laufen. Wichtig ist lediglich, dass die Rechner zusammen genügend RAM und CPU-Leistung besitzen, um die Anzahl gleichzeitiger User simulieren zu können, die für den jeweiligen Testfall gefordert sind.
- Die JEE Anwendung wird verteilt auf zwei JBoss 4.0.3SP1 Instanzen (Hector Core und Hector Connectivity entsprechend) mit je initial 1 GByte Heap unter JRockit 5 auf SuSE Enterprise 9 (Kernel 2.6.5, 32-bit) getestet. Hardware ist eine 2 CPU XEON Maschine mit 4 GByte RAM;
- Die Datenbank ist eine Oracle 10.1.0.4 Instanz auf einem 2 CPU Itanium HP-UX 11.23 System mit 32 GByte RAM mit 1 TByte über SAN angebundenem Storage

Der Testclient ist eine Java-Anwendung, die unterschiedliche Agenten und Tasks mit angegebenen Wahrscheinlichkeiten startet:

- o EarnLoyaltyPointsAgent
 - EarnLoyaltyPointsWorkflowTask (100%)
 - SetMembershipStatusTask (14%)
- o EnrollCustomerAgent
 - EnrollCustomerTask
- o UserAgent

- AuthenticateWorkflowTask (100%)
- GetAccountBalanceDetailsTask (100%)
- GetAccountStatementTask (2.7%)
- RedeemPointsTask (1%)

Für jeden Testlauf wurden folgende Schritte durchgeführt:

- ein Datenbankschema auf die Datenbankserver eingespielt;
- die Datenbank unter Einsatz von einer Java-Anwendung, die dieselbe Persistenzschicht verwendet wie das Hector Core, mit Testdaten beladen;
- die Mapping-Dateien des Hector Core Servers wurden aktualisiert, um die Anbindung des Hector Core an die aktuelle Datenbank herzustellen;
- die Hector Core und Hector Connectivity gestartet und der Testclient aufgerufen.

Beim Aufruf des Testclients werden die Anzahl der gewünschten Threads und die Anzahl der Aufrufe pro Sekunde als Parameter angegeben. Damit kann der Tester die Anzahl der parallelen Zugriffe an die Datenbank steuern.

Als Ergebnis eines Testlaufs bekommt man eine ASCII-Datei mit den Ausführungszeiten pro Methodenaufruf. Ein Beispiel für Einträge in dieser Logdatei:

(Datum; Status; Methode; Identifikationstyp und Nummer;

Ausführungszeit dieses Aufrufs; durchschnittliche Ausführungszeit aller Aufrufe dieser Methode)

```
2007-06-17 20:50:21,853;INFO ;EnrollCustomer;CARD;1234560210194473;153;246
2007-06-17 20:50:21,935;INFO ;AuthenticateWorkflow;CARD;1234560210270030;24;35
2007-06-17 20:50:21,950;INFO ;GetAccountBalanceDetails;CARD;1234560210270030;14;14
2007-06-17 20:50:21,978;ERROR;RedeemPoints;CARD;1234560090129462;53;53
2007-06-17 20:50:22,034;INFO ;AuthenticateWorkflow;CARD;1234560157500539;41;35
```

Eine statistische Bearbeitung dieser Datei liefert einige charakteristische Zahlen wie die durchschnittliche Ausführungszeit der Methoden, die Mediane und, was für praktische Zwecke am wichtigsten ist, die 96%-Latenzzeit (die minimale Zeit, die bei 96% aller Aufrufe nicht überschritten worden ist).

Es sind 2 Sätze von Tests durchgeführt worden: Im ersten Testsatz wurden die Datenbanken an einem Oracle-RAC Cluster bereitgestellt, was neben der Parallelisierung der Datenbankzugriffe auch die Benutzung einer wesentlich größeren Testdatenmenge ermöglicht hat. In diesem Testsatz wurde das Hector Core DB-Schema mit dem DA-DB Schema „Table per hierarchy“ verglichen. Im zweiten Satz sind die Datenbankschemas von Hector Core Datenbank und die beiden DA-Datenbankschemas miteinander verglichen worden. Dies wurde in der oben beschriebenen Testumgebung durchgeführt, am Datenbankserver „SPGHT1“.

Die Testdatensätze, mit denen die Datenbanken beladen worden sind, sieht so aus:

RAC-Cluster – 4000000 Mitglieder, d.h. 4 Mio. Einträge in der Tabellen „Account“, „Membership“ und „Alias“ jeweils. Für jedes Mitglied sind auch 3 Events in die Datenbank eingetragen, d.h. jeweils 12 Mio. Einträge in den Tabellen „Event“, „Transaction“ und „BalanceDetails“.

SPGHT1 – 1 500 000 Mitglieder, jeweils mit 4 Events. Dementsprechend wurden 1,5 Mio. Einträge in der Tabellen „Account“, „Membership“ und „Alias“ angelegt und 6 Mio. Einträge in den Tabellen „Event“, „Transaction“ und „BalanceDetails“.

Dabei wurden auch andere Tabellen mit einer riesigen Datenmenge belegt, aber die Tabellen gehören nicht zur LSD-Komponente und sind deswegen vom geringeren Interesse.

Wegen der betrieblichen Zeitplanung mussten die Tests an dem RAC-Cluster auf eine relativ einfache, oben beschriebene Weise durchgeführt werden. Im Unterschied dazu waren die Tests am SPGHT1-Server vom Ablauf her komplizierter: Erstens, wurden pro DB-Schema jeweils zwei Testläufe nacheinander durchgeführt. Damit kann man mit stabileren und realitätsnäheren Ergebnissen rechnen. Zweitens, wurden vor jedem Testlauf die Oracle Statistiken mit CBO, Cost-Based Optimizer gesammelt. Damit könnte man mit einer Steigerung der Performanz bei der Ausführung von DB-Abfragen rechnen. Und drittens, wurde vor der Testausführung Oracle AWR (Automatic Workload Repository) gestartet. Damit könnte man in der nachfolgenden Analysephase z.B. eventuelle Engpässe und nicht optimisierte Abfragen entdecken. Auf der Basis der von AWR gesammelten Daten hat man außerdem den Oracle ADDM (Automatic Database Diagnostic Monitor) laufen lassen, dessen Reports Anweisungen zur Optimierung der SQL-Abfragen enthalten.

Die Testläufe an dem RAC-Server haben folgende Kennzahlen geliefert:

	1	2	3	4	5	6	7	8
HECTOR								
Average [ms]	261	77	1224	998	104	326	395	230
Median [ms]	54	24	776	558	25	184	123	120
96% [ms]	1274	344	4063	3292	395	1226	1764	738
DA1								
Average [ms]	306	83	1387	1061	104	302	590	279
Median [ms]	51	32	928	519	26	140	170	157
96% [ms]	1484	353	5028	3827	372	1042	2388	779
Differenz, (Hector - DA), %								
Average	-17,24	-7,79	-13,32	-6,31	0,00	7,36	-49,37	-21,30
Median	5,56	-33,33	-19,59	6,99	-4,00	23,91	-38,21	-30,83
96%	-16,48	-2,62	-23,75	-16,25	5,82	15,01	-35,37	-5,56

Die Testläufe an dem SPGHT1-Server haben folgende Kennzahlen geliefert:

96% Latenzzeit, [ms]

	1	2	3	4	5	6	7	8
Hector Testlauf1	525	69	667	742	94	606	525	121
Hector Testlauf2	519	65	645	699	89	782	513	107
DA1 Testlauf1	517	70	643	743	89	612	557	115
DA1 Testlauf2	535	79	682	795	94	638	732	127
DA2 Testlauf1	566	81	715	786	97	645	649	129
DA2 Testlauf2	585	82	703	786	95	638	784	120

Average, [ms]

	1	2	3	4	5	6	7	8
Hector Testlauf1	88	27	150	245	36	175	105	43
Hector Testlauf2	86	27	145	240	35	208	97	40
DA1 Testlauf1	91	27	149	250	37	186	104	42
DA1 Testlauf2	90	28	157	254	37	209	117	44
DA2 Testlauf1	95	29	156	249	40	192	116	46
DA2 Testlauf2	109	30	162	248	37	239	155	44

Median, [ms]

	1	2	3	4	5	6	7	8
Hector Testlauf1	32	14	63	153	20	116	37	28
Hector Testlauf2	32	14	61	153	20	138	35	26
DA1 Testlauf1	32	13	62	156	19	126	35	28
DA1 Testlauf2	32	14	65	155	20	154	37	27
DA2 Testlauf1	32	14	62	147	20	130	37	29
DA2 Testlauf2	33	14	64	149	20	174	38	27

1 - AuthenticateWorkflow, 2 - CheckAlias, 3 - EarnLoyaltyPointsWorkflow, 4 -EnrollCustomer, 5 - GetAccountBalanceDetails, 6 – GetAccountStatement, 7 - RedeemPoints, 8 – SetMembershipStatus

DA1 – DA-Datenbank „Table per hierarchy“, DA2 – DA-Datenbank „Table per subclass“

Die aufgeführten Daten zeigen relativ geringe Abweichungen der Testergebnisse voneinander. Damit kann man mit einer gewissen Sicherheit behaupten, dass die Unterschiede an dem Hector Core Datenbankschema lediglich einen geringen Einfluss auf die Performanz des gesamten Hector Systems haben.

Die Analyse der AWR- und ADDM-Reports bietet auch kein Verbesserungspotential an: die Abfragen, die am langsamsten von der Datenbank ausgeführt werden, haben eine einfache Struktur und nur deswegen langsam sind, weil sie auf einen großen Datensatz zugreifen (dies sieht man aus der AWR-Reports). Der bedeutendste Vorschlag von ADDM, der zur Performanzsteigerung führen soll, heißt „die Anzahl von Commits reduzieren und größere Transaktionen verwenden“, was natürlich für die Applikation mit hohen Nebenläufigkeitsanforderungen wenig Sinn hat.

Damit heißt das Ergebnis dieser Analyse: das Datenbankschema des Hector Core Systems bietet keine Möglichkeit an, ihren Durchsatz wesentlich zu erhöhen.

Fazit

Der MDA-Ansatz hat sich bei der Entwicklung eines komplexen Softwaresystems als mächtiger und gleichzeitig flexibler Werkzeug erwiesen, der einem Entwickler mehrere Vorteile anbieten kann. Insbesondere sind es hohe Qualität, Konsistenz und Portabilität des erzeugten Codes. Im konkreten Fall eines mit MDA-Werkzeug generierten Datenbankschemas wurde im Laufe der Anfertigung dieser Diplomarbeit festgestellt, dass der MDA-Ansatz es zulässt, die zu erzeugende Struktur einer Datenbank mit hoher Präzision zu kontrollieren und einzustellen. Daher hängt die Qualität des erzeugten Datenbankschemas vorrangig von dem Wissen und Erfahrung eines Entwicklers ab.

Literatur

1. Christian Bauer, Gavin King. Hibernate in Action. Manning Publications, 2005
2. Christian Bauer, Gavin King. Java Persistence with Hibernate. Manning Publications, 2007
3. Hibernate Reference Documentation, Version: 3.2.2. <http://hibernate.org/>
4. Eric J. Naiburg, Robert A. Maksimchuk. UML for Database Design. Addison-Wesley, 2001
5. Thomas Kyte. Expert One on One Oracle. Apress, 2003
6. Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions. Apress, 2005

Anhang

Description of the entities of the LSD component.

Account

The Account contains all known balances and works as maintainer/container of Transactions. A member collects units to an account and spends units from an account. It handles all changes to the account balances and uses transactions for the historicizing of those changes.

Attributes

Name	Type	Initial Value	Multiplicity
active	Boolean	TRUE	
lastStatement	DateTime		0..1

active

Defines if the Account is active (or not locked) or not.

Setting an Account to inactive (active=false) prevents any alias spending or collecting to this Account from authenticating.

lastStatement

The lastStatement date defines the point in time when the last time an account statement of new Transactions was requested by the user.

Relations

Name	Type	Begins	Ends
	association	Membership	Account
	association	Account	BalanceDetails
	association	Account	Program
	association	Account	Account
	association	HistoricizedAccount	Account
	association	Account	HistoricizedAccount
bookedOn	association	Account	Transaction

Tagged Values

Tag Definition/Tag Name	Value
Tag Definition: @andromda.hibernate.version : String	modCount

Alias

Direct Subclassifiers:

AssignedAlias, GeneratedAlias

The alias identifies the acting member. It represents the physical identification method of a customer.

Attributes

Name	Type	Initial Value	Multiplicity
active	Boolean	Boolean.TRUE	0..1
code	String (60)		
pin	String (4)		

active

Defines if the alias is active (= not locked) or not. Setting an alias to inactive (active=false) prevents it from authenticating.

code

The aliasCode identifies the alias within a special program and with a special aliasType.

pin

The pin is used for authentication of the customer using an alias.

Relations

Name	Type	Begins	Ends
	association	TransferUnitsEvent	Alias
	association	Event	Alias
	association	TransferUnitsEvent	Alias
	association	Alias	Membership
	association	Alias	AliasType
	association	Alias	Program
	generalization	AssignedAlias	Alias
	generalization	GeneratedAlias	Alias

Tagged Values

Tag Definition/Tag Name	Value
Tag Definition: @andromda.hibernate.version : String	modCount

AliasRange

A successive collection of aliases of particular type and belonging to a particular partner. Aliases are represented via their IDs. The aliases from an alias range are used for the enrollment. Once the alias has been used, it can not be obtained from the range any more. AliasRange maintains its aliases remembering last enrolled, remaining not enrolled aliases, etc. To allow enrollments for a partner, alias ranges of necessary alias types must be created for this partner.

Attributes

Name	Type	Initial Value	Multiplicity
aliasCounter	Integer		
brandingOrganization	String (64)		
endAliasCode	String (60)		
lastCreatedForAliasType	Boolean		
remainderCapacity	Integer		
size	Integer		
startAliasCode	String (60)		
startAliasCodeSequence	Long		
modCount	String		

aliasCounter

Number of already enrolled aliases.

brandingOrganization

partner for whom the aliasRange is allocated

endAliasCode

Code of the alias which is ending the aliasRange - the counting is represented by the increasing of the sequence number which is part of the aliasCode

lastCreatedForAliasType remainderCapacity

number of not yet enrolled aliases.

size

Number of aliases which are contained in the aliasRange

startAliasCode

Code of the alias which is starting the aliasRange - the counting is represented by the increasing of the sequence number which is part of the aliasCode

startAliasCodeSequence

alias code sequence used as a starting point for producing alias codes from the alias range. Obtained from AliasType.

Relations

Name	Type	Begins	Ends
	association	AliasRange	GeneratedAliasType
	association	AliasRange	Program

Tagged Values

Tag Definition/Tag Name	Value
Tag Definition: @andromda.hibernate.version : String	modCount

AliasType

LookupValuePersistent

|

+--AliasType

Direct Subclassifiers:

AssignedAliasType, GeneratedAliasType

This general Type is the superclass for its specializations GeneratedAliasType and AssignedAliasType. They both have an AliasType key in common. With this entity one is also able to find all kind of aliasTypes for the program needed

Relations

Name	Type	Begins	Ends
	association	Alias	AliasType
	generalization	AliasType	LookupValuePersistent
	generalization	AssignedAliasType	AliasType
	generalization	GeneratedAliasType	AliasType

AssignedAlias

Alias

|

+--AssignedAlias

An assigned Alias can be chosen by the customer himself. Depending on the aliasType a user can use something like phonenumber or email-adress as his own aliasId, representing his alias. It is not unique in the system but only one alias with the same aliasId can be "blocked" in the system to make it possible to choose a phonenumber i.e. twice in the system, when another customer gets the phonenumber after the old one finished with the phone company.

Attributes

Name	Type	Initial Value	Multiplicity
blocked	Boolean	Boolean.TRUE	0..1

blocked

this boolean assures, that only one user can actively use the alias which exist be more than one times in the system. Only after deblocking the alias a new alias can be assigned to the system with the same type and aliasId. After assignment the alias is blocked automatically. It assures, that former data are consistent in the system and assigned aliases can be used by other users in the future, to assure a customer friendly attitude

Relations

Name	Type	Begins	Ends
	generalization	AssignedAlias	Alias

AssignedAliasType

AliasType

|

+--AssignedAliasType

With this aliasType one is able to define any format and type for his new alias (i.e. phonenumber, creditcardnr. ...). One just needs to define a regular expression for the format of the new aliasType which guarantees the correctness of the aliasId which is assigned by the user himself.

Attributes

Name	Type	Initial Value	Multiplicity
pattern	String (150)		

pattern

The pattern is a regular expression which will validate the format of any aliasId that the user wants to assign for this specific type

Relations

Name	Type	Begins	Ends
	generalization	AssignedAliasType	AliasType

BalanceDetails

Class to hold all amounts for a virtual currency. For every virtual Currency defined for a program a balanceDetail exists for every account. You can collect units and spend units and the collected can be blocked from the beginning. Blocked units will be unblocked when their blockedUntil time is over. This is done automatically from within the account by calling the checkBlocked routine (which is done every time when a balance is requested).

Attributes

Name	Type	Initial Value	Multiplicity
balance	VirtualCurrencyQuantity		
blocked	VirtualCurrencyQuantity		
dueToExpire	VirtualCurrencyQuantity		
firstNotExpiredTrx	DateTime		0..1
lastExpired	VirtualCurrencyQuantity		
lastExpiryRun	DateTime		0..1
nextUnblockTime	DateTime		0..1
totalCollected	VirtualCurrencyQuantity		
totalSpend	VirtualCurrencyQuantity		
virtualCurrency	VirtualCurrency		

balance

The amount of units that the account holds for the virtual currency. This is not the spendable amount. The spendable amount does not contain blocked units, which this amount does.

blocked

The number of units currently blocked.

dueToExpire

Amount that is due to expire at the next units expiry run.

firstNotExpiredTrx

This is the effectiveTime of the oldest (first) not expired transaction. This is used for units expiry to easily find out if the account has transactions that could possibly expire. If this is null, no transactions have been booked on the account or all have expired / been spended. Account does not have units possible to expire when this date is null or this date is younger than the expiry date.

lastExpired

Amount that had expired at the last units expiry run.

lastExpiryRun

The date of the last expiry run that changed this balance. An Expiry run takes some time to change all accounts. Therefore this date could be from the run before the actual run. If this is the case the dueToExpire units are actually already expired (that what normally would be in lastExpired). The lastExpired are from the penultimate run and no longer interesting.

nextUnblockTime

Contains the date when the next time an amount is unblocked for this balance. When getting the balance this date is checked and only if it is past the blocked amount is recalculated.

totalCollected

The number of totally collected units since the creation of the account

totalSpend

The total amount of units spended since the creation of the account.

virtualCurrency

Virtual Currency of this balanceDetails. All units are of this currency type. Every account has as many balanceDetails as virtualCurrencies are defined for the affected program.

Relations

Name	Type	Begins	Ends
	association	BlockedTransaction	BalanceDetails
	association	Account	BalanceDetails

Tagged Values

Tag Definition/Tag Name	Value
Tag Definition: @andromda.hibernate.version : String	modCount

BlockedTransaction

Holds all currently blocked transactions. Each blocked transaction is assigned to one balance and to one transaction. When the blockedUntil date is over and the next access to the account had been taken place, this entry gets deleted (and the units unblocked). It would have been possible to hold this value as an additional field in the transaction. Since the blocked transactions are only a small fraction of all transactions and a transaction shall not be changable this is put into an own object.

Attributes

Name	Type	Initial Value	Multiplicity
------	------	---------------	--------------

blockedAmount	VirtualCurrencyQuantity		
blockedUntil	DateTime		

blockedAmount

The amount of units that are blocked. This is normally the whole amount of units of the transaction, but can also be fractions of it.

blockedUntil

The date / point in time until the units are blocked.

Relations

Name	Type	Begins	Ends
	association	BlockedTransaction	BalanceDetails
	association	BlockedTransaction	Transaction

CodeFormat

Represents a format for an identifier e.g. the aliasId. Identifier represented by a string may be generated or checked against the format.

Attributes

Name	Type	Initial Value	Multiplicity
checksumNeeded	Boolean		
completeLength	int		
prefix	String (30)		

Attributes

Name	Type	Initial Value
checksumNeeded	Boolean	
completeLength	int	
prefix	String	0..1

checksumNeeded

This boolean allows to define whether a checksum is added at the end of the aliasId for further or later checks of the aliasId

completeLength

The number of figures which are allowed for the generated aliasId. The figures which are not filled by the prefix, sequence number or checksum figures are filled with nulls.

prefix

An optional prefix which allows a better overview and sorting for the generated aliasIds

CompositeEvent

Event

|

+--CompositeEvent

Defines an Event that contains various other Events. not used...

Relations

Name	Type	Begins	Ends
------	------	--------	------

	association	CompositeEvent	Event
	generalization	CompositeEvent	Event

EkitTypeAliasTypeMapper

Relations			
Name	Type	Begins	Ends
	association	EkitTypeAliasTypeMapper	GeneratedAliasType
	association	EnrollmentKitType	EkitTypeAliasTypeMapper

EnrollmentEvent

Event

|

+--**EnrollmentEvent**

The EnrollmentEvent signals an enrollment and can be used to trigger the rewarding of units for new member.

Relations			
Name	Type	Begins	Ends
	generalization	EnrollmentEvent	Event

EnrollmentKit

A collection of aliases given to a customer for participating in a program. There is only one account for all aliases from the kit.

Attributes			
Name	Type	Initial Value	Multiplicity
brandingOrganization	string(64)		1

Relations			
Name	Type	Begins	Ends
	association	EnrollmentKit	EnrollmentKitType
	association	EnrollmentKit	Program
	association	EnrollmentKit	GeneratedAlias

brandingOrganization

unique identifier (ldapId) of the partner the ekit is created for

EnrollmentKitType

LookupValuePersistent

|

+--**EnrollmentKitType**

Defines the types of the aliases and the roles of their memberships in an enrollment kit. Note: there is only one account per all aliases/memberships in an enrollment kit.

Attributes			
Name	Type	Initial Value	Multiplicity
brandingOrganization	string(64)		0..1

brandingOrganization

the unique ldap id of the organization the ekitType is created for. If this is not set the ekitType is program wide valid

Relations

Name	Type	Begins	Ends
	association	EnrollmentKitType	EkitTypeAliasTypeMapper
	association	EnrollmentKit	EnrollmentKitType
	generalization	EnrollmentKitType	LookupValuePersistent

Event

Direct Subclassifiers:

EnrollmentEvent, UnitsExpiryEvent, CompositeEvent, TransferUnitsEvent, SpendEvent

The Event is a business action that can result into units awarded to the member.

Attributes

Name	Type	Initial Value	Multiplicity
blockedUntil	DateTime		0..1
effectiveTime	DateTime		
location	string(64)		
noticedTime	DateTime		
type	int		

blockedUntil

The units awarded are blocked (not available to the customer) until this point in time.

effectiveTime

Point in time when the event occurred. Relevant timestamp for rating.

location

unique identifier (ldap id) of the partner (businessOrganization) the event has taken place which is important information for clearing

noticedTime

Point in time when the event was taken over by the hector system. Ideally effectiveTime, noticedTime and insertedAt refer to the same point in time but there exist several scenarios in which this constraint can not be ensured. In case of batch processing the noticedTime is the creation time of the file containing the event. The hector system is responsible to process the event as soon as possible and to insert it in the database (insertedAt). However especially in batch processing there is a gap between these timestamps. But even in the case of online processing it can not be ensured by the hector system that the event is delivered as soon as it occurs. Relevant timestamp for partner clearing.

type

The type of the event.

Relations

Name	Type	Begins	Ends
	association	Transaction	Event
	association	CompositeEvent	Event
	association	Event	Program
	association	Event	Alias
	generalization	EnrollmentEvent	Event
	generalization	UnitsExpiryEvent	Event
	generalization	CompositeEvent	Event

	generalization	TransferUnitsEvent	Event
	generalization	SpendEvent	Event

ExpiryRun

Holds information about an expiry run which is all information that can be configured for units expiry. Information about the running job itself like percentage completed or other monitoring values can be found in the job framework.

Attributes

Name	Type	Initial Value	Multiplicity
displayPeriod1End	DateTime		0..1
displayPeriod1Start	DateTime		0..1
displayPeriod2End	DateTime		0..1
displayPeriod2Start	DateTime		0..1
goodwillPeriodEnd	DateTime		
status	string(30)	ExpiryRunStatusType.CREATED	
ueCutOffDate	DateTime		
ueEffectiveDate	DateTime		

displayPeriod1End

End of first period of special information displayed

displayPeriod1Start

Start of first period of special information displayed

displayPeriod2End

End of second period of special information displayed

displayPeriod2Start

Start of second period of special information displayed.

goodwillPeriodEnd

The date until goodwill units can be given. The goodwill period starts at effectiveDate

status

The status of the expiry run.

ueCutOffDate

The point in time where all transactions that were booked before will expire (if they are not spent of course). Its normally UEEffectiveDate - UnitLifeSpan.

ueEffectiveDate

The exact date when the due to expire units will expire. After that date the expired units are no longer available to the customer, but in the beginning they are still not physically expired. The job of physically expiring takes a longer time, but the expiration must take effect at one specific point in time.

Relations

Name	Type	Begins	Ends
	association	UnitsExpiry	ExpiryRun
	association	UnitsExpiry	ExpiryRun
	association	UnitsExpiry	ExpiryRun
	association	UnitsExpiry	ExpiryRun

Fraud

Attributes

Name	Type	Initial Value	Multiplicity
reason	String (350)		

reason

Relations

Name	Type	Begins	Ends
	association	Fraud	SpendEvent

FraudCheckingRuleViolation

Attributes

Name	Type	Initial Value	Multiplicity
description	String (250)		
ruleName	String (60)		

description

ruleName

Relations

Name	Type	Begins	Ends
	association	FraudSuspicion	FraudCheckingRuleViolation

FraudSuspicion

Relations

Name	Type	Begins	Ends
	association	FraudSuspicion	SpendEvent
	association	FraudSuspicion	FraudCheckingRuleViolation

GeneratedAlias

Alias

|

+--GeneratedAlias

A generated Alias is provided by the system and the user just wants to get an alias of a generated type. An aliasId will be generated by the system and one is able to reprint the alias. The enrollment of a customer needs a generated alias, it can not be initially enrolled by an assigned alias.

Attributes

Name	Type	Initial Value	Multiplicity
brandingOrganization	string(64)		
lastReprint	DateTime		

brandingOrganization

The unique identifier (ldap style) of the organization the customer the alias is branded of.

lastReprint

Date of the lastReprint to find the new date for reprint according to the reprint frequency

Relations			
Name	Type	Begins	Ends
	association	EnrollmentKit	GeneratedAlias
	association	GeneratedAlias	Program
	generalization	GeneratedAlias	Alias

GeneratedAliasType

AliasType

|

+--GeneratedAliasType

This alias is the mostly used aliasType (i.e. enrollment, defaultAliasType, aliasRanges ...) It is generated by the hector system and defined by a special IDFormat. When a new alias is requested because of enrollment i.e. than depending on the partners aliasRange and the format of this aliasType a new Aliasid will be generated

Attributes			
Name	Type	Initial Value	Multiplicity
aliasCodeFormat	CodeFormat		
aliasCounter	Long		0..1
reprintFrequency	string(30)		
validForEnrollment	Boolean		

aliasCodeFormat

The code format which is needed to generate a unique aliasCode for the given type. It consists of a prefix which is optional, a sequence number, the aliasCounter starts at and a boolean value to decide wheter a checksum part is added at the end of the aliasCode

aliasCounter

This intern counter allows the aliasRange to avoid duplicates of aliasIds

reprintFrequency

This frequency manages the reprint run of a generated alias.

validForEnrollment

Relations			
Name	Type	Begins	Ends
	association	Program	GeneratedAliasType
	association	EkitTypeAliasTypeMapper	GeneratedAliasType
	association	AliasRange	GeneratedAliasType
	generalization	GeneratedAliasType	AliasType

HistoricizedAccount

Attributes			
Name	Type	Initial Value	Multiplicity
historicizationDate	DateTime		

historicizationDate

Relations			
Name	Type	Begins	Ends
	association	HistoricizedAccount	Account
	association	Account	HistoricizedAccount
	association	Membership	HistoricizedAccount

HistorizedMembershipStatus

Attributes			
Name	Type	Initial Value	Multiplicity
before	Date		0..1
from	Date		

before
from

Relations			
Name	Type	Begins	Ends
	association	Membership	HistorizedMembershipStatus
	association	HistorizedMembershipStatus	MembershipStatus

Tagged Values

Tag Definition/Tag Name	Value
Tag Definition: @andromda.hibernate.version : String	modCount

LookupValue

Attributes			
Name	Type	Initial Value	Multiplicity
key	String		
type	String		

key
type

LookupValuePersistent

Direct Subclassifiers:

MembershipStatus, EnrollmentKitType, AliasType

This class is the superClass for all lookupValues within the system. They are used to provide the opportunity of creating dynamic values for every program which will be lookedUp in the database always in the same way whenever needed by the system.

Attributes			
Name	Type	Initial Value	Multiplicity
key	String (60)		

key

The key of the lookup value must be unique within the given program and the several subtype. This will be assured within the create-methods of every subclass

Relations			
Name	Type	Begins	Ends
	association	Program	LookupValuePersistent
	generalization	MembershipStatus	LookupValuePersistent
	generalization	EnrollmentKitType	LookupValuePersistent
	generalization	AliasType	LookupValuePersistent

Membership

The Membership connects the real person (customer) to a program. The membership is identified through the alias.

Attributes

Name	Type	Initial Value	Multiplicity
customerid	String (80)		0..1
role	int		

customerid

Link of the membership to the customer. The customer must already exist when this connection is made. As long as the customer is not enrolled to a membership and this connection isn't existing, the account is anonymous and the users only can collect units.

role

Relations			
Name	Type	Begins	Ends
	association	Membership	Account
	association	TermsOfAgreement	Membership
	association	Alias	Membership
	association	Membership	Program
	association	Membership	HistorizedMembershipStatus
	association	Membership	HistoricizedAccount

Tagged Values

Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

MembershipService

A membershipService is created for a program and a channel within a given time period. Every customer who has booked this special service will get information within this timePeriod with the given frequency using the given channel.

Attributes

Name	Type	Initial Value	Multiplicity
channel	string(30)		
endTime	DateTime		0..1
frequency	string(30)		
name	String (60)		
startTime	DateTime		

channel

Channel of the membershipService (e.g. Email, SMS, letter a.s.o)

endTime

Point in time when the time period of the membershipService shall end

frequency

The frequency with which the service is repeated during its validity time period

name

Name of the membershipService

startTime

Point in time when the time period of the membershipService shall start

Relations			
Name	Type	Begins	Ends
	association	TermsOfAgreement	MembershipService
	association	MembershipService	Program

Tagged Values		
Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

MembershipStatus

LookupValuePersistent

|

+--MembershipStatus

The MembershipStatus holds the logic of the statusConcept. In the future thresholds and limits can be defined beside the already existing key which is a lookupvalue key

Relations			
Name	Type	Begins	Ends
	association	HistorizedMembershipStatus	MembershipStatus
	generalization	MembershipStatus	LookupValuePersistent

Tagged Values		
Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

Program

The program is the main entity of the system. Nearly every part of hector is program dependent. Without an existing and totally configured program no transactions and no events can be triggered. The name is unique and can only be defined once.

Attributes			
Name	Type	Initial Value	Multiplicity
name	String(60)		
refCurrency	string(3)		0..1

name

The name of the program is the unique key within the whole system.

refCurrency

The reference currency of the program. It will be used as reference currency system wide. TODO: Where will it be used?

Relations

Name	Type	Begins	Ends
	association	Program	UnitsExpiry
	association	Program	LookupValuePersistent
	association	Program	RuleSet
	association	Program	GeneratedAliasType
	association	VirtualCurrencyConfig	Program
	association	Converter	Program
	association	AliasRange	Program
	association	EnrollmentKit	Program
	association	Event	Program
	association	Account	Program
	association	MembershipService	Program
	association	Membership	Program
	association	Alias	Program
	association	GeneratedAlias	Program

Tagged Values

Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

SpendEvent

Event

|

+--SpendEvent

Special event which is processed when customer wants to spend his/her units

Attributes

Name	Type	Initial Value	Multiplicity
amount	VirtualCurrencyQuantity		

amount

amount that are wanted to be spend. Balance details will be checked if there are enough units to spend.

Relations

Name	Type	Begins	Ends
	association	FraudSuspicion	SpendEvent
	association	Fraud	SpendEvent
	generalization	SpendEvent	Event

TermsOfAgreement

Each term of agreement covers a component of the contract which represents the booking of a membership service by a membership. The terms of agreement consist of the channel, which is used to determine the details, a valid period and a property list, which contains the details of the terms of agreement. Additionally each term connects exactly one Membership to exactly one MembershipService.

Attributes

Name	Type	Initial Value	Multiplicity
channel	string(30)		
endTime	DateTime		
startTime	DateTime		

channel

The channel which was used to determine the term of agreement.

endTime

The end time of the valid period for the terms of agreement.

startTime

The start time of the valid period for the terms of agreement.

Relations

Name	Type	Begins	Ends
	association	TermsOfAgreement	Membership
	association	TermsOfAgreement	MembershipService
	dependency	TermsOfAgreement	Channel

Tagged Values

Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

Transaction

The transaction defines a change of one balance of the assigned account. It serves as balance change history. The account balance a transaction belongs to is retrieved by the currency. There can only be one balance for one currency for one account.

Attributes

Name	Type	Initial Value	Multiplicity
amount	VirtualCurrencyQuantity		
effectiveTime	DateTime		
type	int		

amount

The amount of units added / subtracted (if negative) to the account balance.

effectiveTime

Point in time when the source event occurred. The same data as in the event itself, only copied here for performance purposes.

type

The type of the transaction can either be COLLECT(1) or SPEND(2). This could also be calculated by checking if the amount is positive (COLLECT) or negative (SPEND). For performance reasons this is pre-calculated.

Relations			
Name	Type	Begins	Ends
	association	Transaction	Event
	association	BlockedTransaction	Transaction
bookedOn	association	Account	Transaction

Tagged Values

Tag Definition/Tag Name	Value	Documentation
Tag Definition: @andromda.hibernate.version : String	modCount	

TransferUnitsEvent

Event

|
+--TransferUnitsEvent

Event for transferring units from one account to another. @deprecated: not used anymore

Attributes

Name	Type	Initial Value	Multiplicity
amount	VirtualCurrencyQuantity		

amount

The amount of units transfered from one account to the other

Relations

Name	Type	Begins	Ends
	association	TransferUnitsEvent	Alias
	association	TransferUnitsEvent	Alias
	generalization	TransferUnitsEvent	Event

UnitsExpiry

The master data for units expiry. This is mainly configuration. It also contains links to a special set of expiry run information.

Attributes

Name	Type	Initial Value	Multiplicity
unitLifeSpan	VirtualCurrencyQuantity		

unitLifeSpan

The standard age in months of units before they expire. Can range from one month to a maximum of 10 years (120 month).

Relations

Name	Type	Begins	Ends
	association	UnitsExpiry	ExpiryRun
	association	UnitsExpiry	ExpiryRun
	association	UnitsExpiry	ExpiryRun

	association	UnitsExpiry	ExpiryRun
	association	Program	UnitsExpiry

UnitsExpiryEvent

Event

|

+--**UnitsExpiryEvent**

An Event specifying that units where expired for this balance.

Attributes

Name	Type	Initial Value	Multiplicity
expiredVirtualCurrencies	VirtualCurrencyQuantity		0..1

expiredVirtualCurrencies

The amount of expired units. This value will be used to calculate the amount for the expiry transaction.

Relations

Name	Type	Begins	Ends
	generalization	UnitsExpiryEvent	Event