Data Warehouse from Scratch – mit ODI 12*c*

Bernhard Rosenberger, MT AG Frankfurt

Oracle hat mit dem Data Integrator 12c (ODI) ein vielversprechendes ETL-Werkzeug als OWB-Nachfolger im Portfolio. Kann der ODI den Anforderungen an ein auf der grünen Wiese zu errichtendes DWH genügen? Und wird er in der aktuellen Version 12c die Nachfolge-Versprechungen gegenüber dem guten alten Warehouse Builder (OWB) erfüllen? Ein Projektbericht.

Der Kunde des Autors, die GFKL Financial Services AG, ist einer der führenden Finanzdienstleister für Forderungsmanagement. Das Unternehmen ist aufgrund einer großen Vielfalt zu verwaltender Einzeltransaktionen auf effizient strukturierte Datenhaltung angewiesen, um eine kundenorientierte, schlanke Abwicklung seines Geschäfts sicherstellen zu können. Ganz oben auf der Agenda des Kunden stand die Modernisierung des Berichtswesens. Dieses basierte auf einer in die Jahre gekommenen, historisch gewachsenen heterogenen ETL-Landschaft. Folgende Anforderungen wurden an ein Nachfolgesystem gestellt:

- Zentral organisierte Datenbereitstellung
- Vereinheitlichte Semantik
- Feingranulare Datenablage
- Offene Architektur
- Zukunftsfähige Technologie
- Tägliche Beladung sicherstellen
- Wiederaufsetzbarkeit von ETL-Schritten

Um die IT-Infrastruktur auf lange Sicht strategisch auszubauen und eine flexible Erweiterbarkeit zu ermöglichen, hat sich die GFKL dazu entschieden, ein Data Warehouse als Basis-Plattform für ein unternehmensweites Berichtswesen neu zu entwickeln. Eine der Herausforderungen des Projekts bestand darin, die aus dem breit gefächerten Leistungs-Portfolio des Unternehmens sich ergebende Datenbasis in geeigneter Form vereinheitlicht, aber auch erweiterbar zusammenzuführen.

Tool-Auswahl und Architektur-Entscheidungen

Wegen der im Hause fundiert vorhandenen technischen Kenntnisse und auch der Chance, „auf der grünen Wiese" beginnen zu können, fiel die Entscheidung, ein DWH mit der aktuellen Datenbank-Version 12*c*, dem ODI 12*c* als ETL-Werkzeug, dem SQL Developer und Apex zur Stammdaten-Pflege komplett neu aufzubauen. Zur Unterstützung in Architekturfragen und bei der Realisierung wurde die MT AG mit hinzugezogen. Ein gemeinsam durchgeführter Proof of Concept mit BI-Tool-Anbietern destillierte MicroStrategy als Werkzeug der Wahl heraus.

Die fachliche Herausforderung bestand in der semantischen Daten-Integration unterschiedlich ausgerichteter Tochtergesellschaften mit dem Ziel, ein zukunftsfähiges, vereinheitlichtes Berichtswesen mit Dashboards und KPIs für die Geschäftsleitung aufzubauen.

Ziel-Architektur

Die Staging Area unterteilt sich dabei in die klassische „1:1"-Datenübernahme aus den Vorsystemen und einen historisierten Cleansing-Bereich, in dem Datentyp-Konvertierungen und Umschlüsselungen umgesetzt werden *(siehe Abbildung 1)*. Die Historisierung der Daten erfolgte nach der von Ralf Kimball als „SCD2" beschriebenen Methode (Gültigkeit von Datensätzen durch Zeit-Intervall).

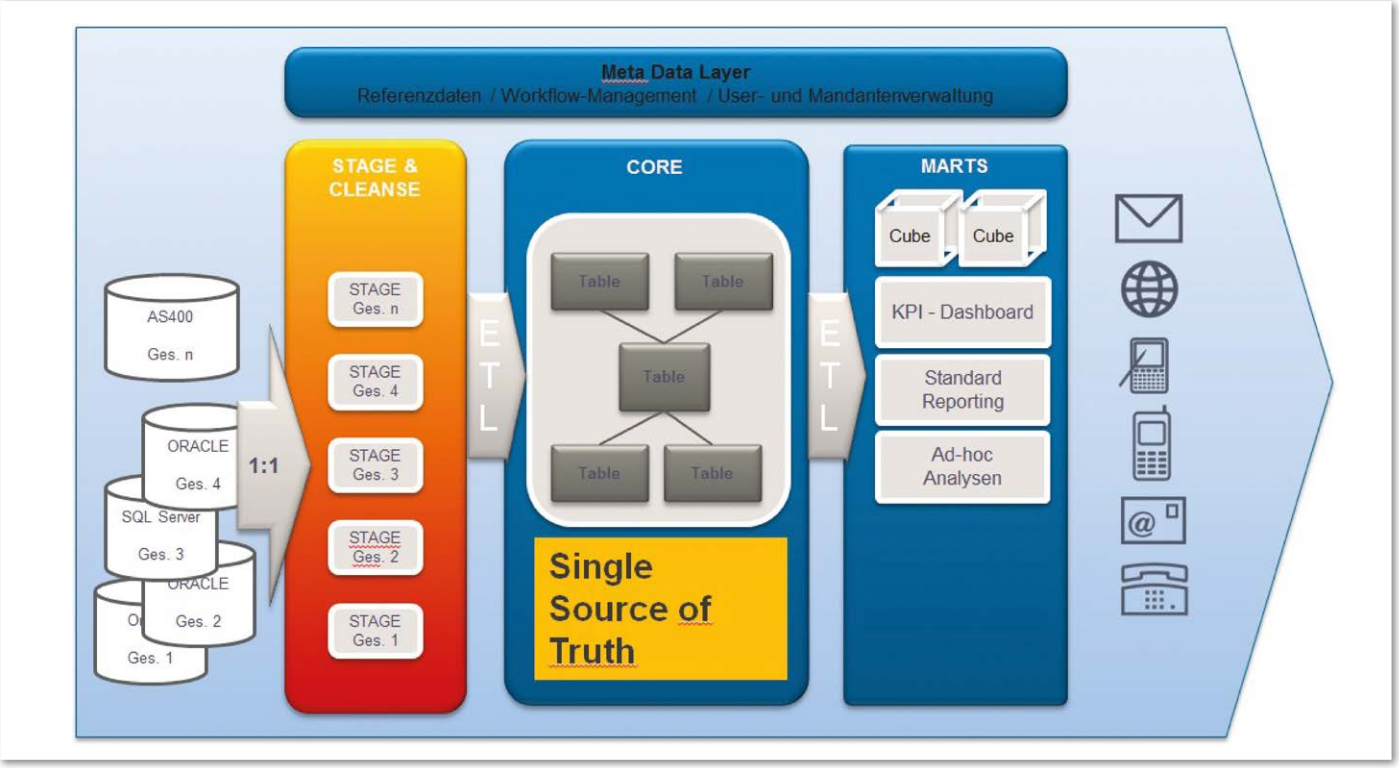

Abbildung 1: DWH Ziel-Architektur

Im Core werden die Daten basierend auf den Ergebnissen der fachlichen Analyse historisiert, strukturiert und integriert gespeichert; sie dienen als Ausgangsbasis zur Erstellung verschiedener Datamarts, die für die jeweiligen Berichte in die Abteilungen und die Geschäftsleitung bereitgestellt werden müssen. Die in ROLAP-Technologie erstellten Datamarts sind als Star- und Snowflake-Konstrukte zur optimierten Verwendung durch das eingesetzte BI-Tool MicroStrategy aufgebaut.

Das Projekt bot die einmalige Chance, die Praxistauglichkeit des ODI 12*c* jenseits von Labor-Bedingungen eingehend auf Herz und Nieren zu prüfen. Die dabei gemachten Erfahrungen sind nachfolgend näher dargestellt und abschließend im Vergleich zum Vorgänger-Tool OWB bewertet.

Projekt-Methodik

Das Projekt war gekennzeichnet von kurzen Entwicklungsphasen mit klar definierten Zielen. Das Team setzte sich aus Mitgliedern der Fachbereiche, fachlichen Analysten und Entwicklern zusammen. Tägliche Standups und ein offenes Kommunikationsklima förderten den Informationsaustausch besonders auch auf informeller Ebene und ermöglichten ein schnelles Erkennen und Reagieren auf ungeplanten Handlungsbedarf. Bei aller Agilität gaben definierte Etappenziele dem Team die notwendige Orientierung für ein zielgerichtetes Gelingen des Projekts. Grob skizziert waren die Projektphasen unterteilt in:

- Fachliche Analyse der Geschäftsprozesse und Datenstrukturen
- DWH-Neubau mit semantischer Daten-Integration und Ableitung erster KPI-Reports
- Historisierung des DWH mithilfe der Knowledge-Module
- Einführung des BI-Tools (MicroStrategy), Erstellung von Reports
- Weitere Phasen im Kontext eines regulären Lifecycle-Managements

Installation und Konfiguration der ODI-12*c*-Umgebung

Da das Projekt auf der grünen Wiese gestartet wurde, gab es zu Beginn Überlegungen, wie der ODI sinnvoll aufgesetzt werden kann. Wie soll die Trennung von Entwicklung und Produktion aussehen, welche Repositories sind erforderlich?

Welche ODI-Agents sollen zum Einsatz kommen? Neben dem bewährten Stand-alone-Agent, seit 11c ein JEE-Agent (Betrieb auf WebLogic Server), steht mit der aktuellen 12*c* ein Collocated-Agent (Betrieb auf Cloud Control) zur Auswahl. Die Entscheidung fiel zugunsten der schlanken und auch stabilen Lösung mit dem Stand-alone-Agent, der auch ohne schwergewichtiges Verwaltungswerkzeug auskommt. Mehrere mögliche Repository-Konfigurationen waren zu bewerten, um im Umfeld der GFKL ein Master-Repository mit vier Work-Repositories als sinnvollste Konfiguration zu identifizieren.

Aufbau der Staging Area – Automatisierung mit Groovy

Zum Aufbau des Staging mussten vier Vorsysteme mit jeweils fünfundvierzig Tabellen angebunden werden. Um die unvermeidliche Fleißarbeit bei der Implementierung gleichartiger Mappings zu vermeiden, kam die an Java erinnernde Groovy Scripting Language erfolgreich zum Einsatz. Es reichte aus, eine Beladungsstrategie einmalig (etwa Datapump-Load oder Delta-Abzug) zu skripten, um daraus ODI-Ladestrecken für jede beliebige Tabelle automatisch generieren zu können. Das Vorgehen reduzierte die manuellen Aufwände erheblich und darüber hinaus die Fehlerquote bei der Stage-Implementierung.

In ODI Studio 12*c* ist erstmals eine rudimentäre Groovy-Runtime-Umgebung eingebunden. *Listing 1* zeigt ein Snippet, das die intuitive Erlernbarkeit der Skriptsprache erahnen lässt. Der Code generiert in einem Mapping zwei Eingangsverbindungen eines ODI-Set-Operators. Zum schnellen Einstieg in das Thema „Groovy" gibt es den

```
…
icps = comp_set.getInputConnectorPoints() //2 inp. Conn.points
i = icps.iterator()
icp1 = i.next() // First input connector point
icp2 = i.next() // Second input connector point
comp_1.connectTo(icp1)
comp_2.connectTo(icp2)
…
```

Listing 1

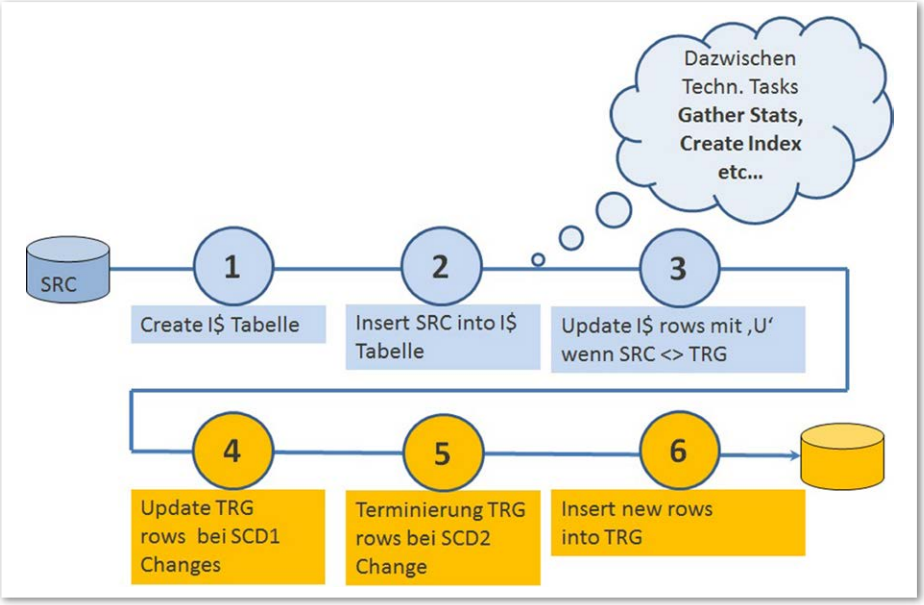

Abbildung 2: SCD2-Knowledge-Modul Kern-Algorithmus

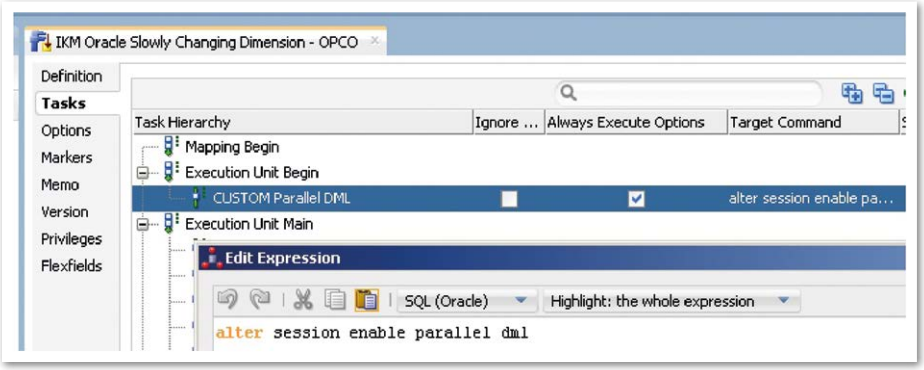

Abbildung 3: KM Custom Task hinzufügen

Oracle-Blog „ODI 12*c* – Mapping SDK the ins and outs" [1].

ODI-Mappings im Core

Im Core waren teilweise komplexe Transformationen durchzuführen. Der ODI unterstützt in 12*c* erstmals neue Operatoren, die zusammen mit den im ODI altbekannten „Set", „Join" und „Filter" eine Realisierung von Transformationen als Mappings á la OWB ermöglichen. Die neu hinzugekommenen Operatoren „Aggregator", „Dataset", „Distinct", „Expression", „Lookup", „Sort" und „Split" basieren auf einem neuen Konzept von sogenannten „Component-Style-Knowledge-Modulen", die zur Laufzeit entsprechend generierte Code-Artefakte beisteuern.

Die komplexeren Mappings wurden aus Zeitgründen zu Beginn basierend auf Datenbank-Views als Datenquelle realisiert, in denen die komplexe SQL-Logik gekapselt ist. Erste Versuche haben gezeigt, dass sich mithilfe der neuen ODI-Operatoren komplexe Mappings ebenso gut realisieren lassen wie mit dem OWB. Damit sind zukünftige View-Kapselungen obsolet. Im ODI-Mapping wären darüber hinaus Objekt-Abhängigkeiten für den Entwickler direkt sichtbar und zusätzlich eine durchgängige Lineage- und Impact-Analyse durch den ODI möglich, die mit dem View-Ansatz unterbrochen wird.

Um die Entwicklungsarbeiten beschleunigt zum Ziel bringen zu können (Quick-Win), ist man beim gekapselten View-Ansatz geblieben, da die View-Darstellung aus den Ergebnissen der Business-Analyse gut abgeleitet werden konnte. Ein Reengineering des Mappings wurde bewusst auf einen späteren Zeitpunkt verschoben.

Knowledge-Module

Als herausragendes Architektur-Feature des ODI stellten sich die Knowledge-Module (KM) heraus. Durch den Einsatz des out of the box mitgelieferten KM für die SCD2-Historisierung konnte man diese Funktionalität im Cleansing und Core enorm schnell prototypisch realisieren. Es genügte, mit Groovy-Skripting ein einfaches Mapping von Quelle zu Ziel zu generieren, diesem das gewünschte KM zuzuweisen und eine SCD-Attribut-Deklaration für Tabellenspalten im Model Editor vorzunehmen. Abschließend lassen sich eventuell noch im Mapping die spezifischen Parameter des KM anpassen. Oracle hat in 12*c* die Bedienbarkeit des Model-Editors verbessert. Nur ein durchaus umschiffbarer GUI-Bug in der Dropdown-Liste für „SCD Behaviour" fiel negativ auf.

Erste Tests lieferten so in Kürze brauchbare Ergebnisse. Um die Historisierung an die Projektbedürfnisse anzupassen, wurden folgende zu ergreifende Tuning-Maßnahmen identifiziert:

- Im Quellsystem gelöschte Datensätze sind im historisierten Ziel zu terminieren
- Die Parallelisierung ist datenbankseitig zu aktivieren, da nicht als Default-Einstellung gegeben („alter session enable parallel DML")
- Intermediär vom KM erzeugte Hilfstabellen sind zu partitionieren
- „Direct Path Loads" sind durch Hints zu aktivieren

Aufgrund des flexiblen wie mächtigen KM-Konzepts waren diese Forderungen durch Anpassungen und Erweiterungen zügig umsetzbar, wie nachfolgend skizziert wird.

Das SCD2-Integration-KM „IKM Oracle Slowly Changing Dimension" sieht out of the box fünfundzwanzig konfigurierbare Execution Unit Tasks vor, die von außen über Parameter (Options) steuerbar sind. Die prinzipielle Funktionsweise lässt sich anhand der sechs wichtigsten Tasks illustrieren *(siehe Abbildung 2)*. Alle weiteren, hier nicht dargestellten neunzehn Tasks sind für technische Schritte wie das Berechnen von Statistiken, das Erzeugen beziehungsweise Abräumen von Hilfstabellen, Indizes etc. notwendig.

Jeder Task im Knowledge-Modul wiederum besteht aus einer Art Skriptcode-Schablone, die zur Laufzeit mit den Metadaten aus dem ODI Repository substituiert wird, um lauffähigen Code zu generieren. Die vier aufgelisteten Anforderungen sind also durch entsprechende Anpassung des Skript-Codes im KM und, wenn nötig, durch das Hinzufügen neuer Mapping-Parameter lösbar.

Nachfolgende Screenshots illustrieren, wie im KM-Editor Tasks für das Terminieren gelöschter Sätze und für das Ein- und Ausschalten der Parallelisierung hinzugefügt werden. Hinzu kamen neue Parameter (im Editor „Options" genannt), um diese Tasks aus jedem Mapping heraus spezifisch steuern zu können *(siehe Abbildungen 3 – 5)*.

Aus dem originalen KM abgeleiteter Code hatte eine sofortige Auswirkung auf alle darauf umkonfigurierten SCD-Mappings. Lediglich eine individuelle Anpassung der Mapping-spezifischen Option ist im Bedarfsfall bei gewünschten Abweichungen vom voreingestellten „DEFAULT"-Wert noch durchzuführen. Alle beschriebenen Maßnahmen halfen, die Laufzeiten der Mappings um ein Vielfaches zu reduzieren. Weiterführende Informationen zum Thema stehen im Blog „Maßgeschneidertes und performantes ETL durch ODI-Knowledge-Modul-Modifikationen" [2].

Abbildung 4: KM Optimizer Hint per Option hinzufügen

```
create table <%=odiRef.getTable("L", "INT_NAME", "A")%> (       <%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] nul
if ( odiRef.getOption( "CUSTOM_PARTIONING_FLOW_TAB" ).equals("TRUE") )
{ %> PARTITION BY LIST(dwh_opco_id) (PARTITION DUMMY VALUES (-1,-2), PARTITION XYA VALUES(1), PARTITION XYB VALUES(2),
```

Abbildung 5: KM Customized Partitioning

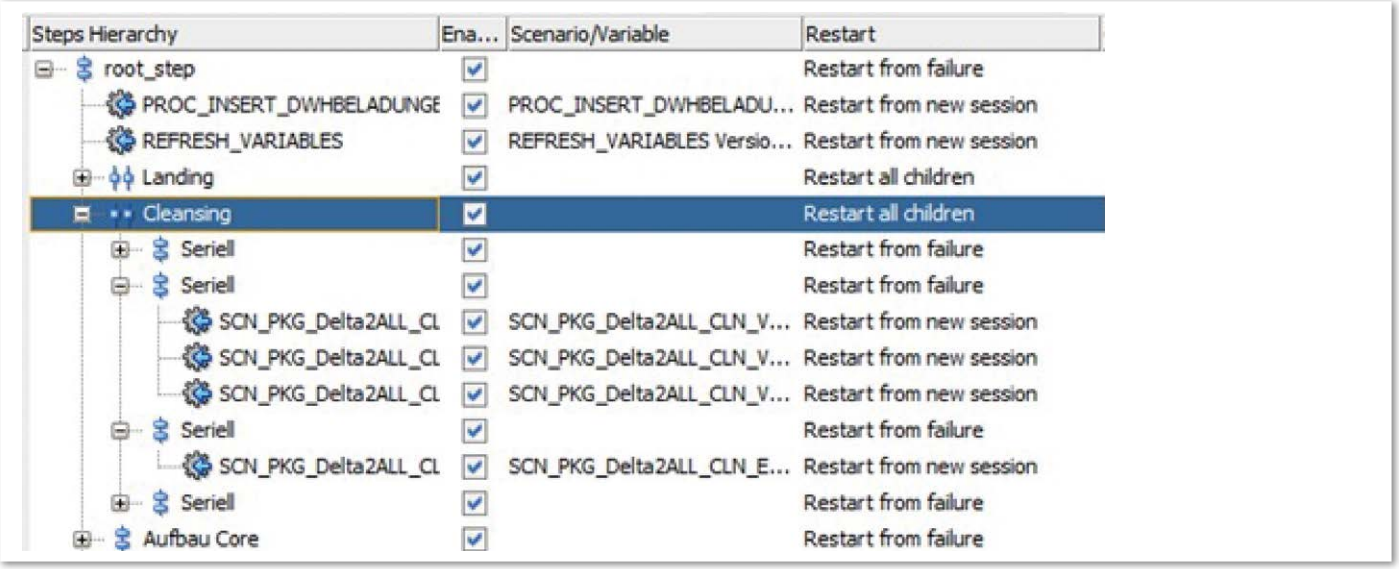

Abbildung 6: Load Plan – Vierfach-Parallelisierung im Cleansing

Packages und LOAD Plans

Dank Packages und LOAD Plans, die seit ODI 11*g* eingeführt worden sind, kann auf den Einsatz eines Drittanbieter- Tools zur Jobsteuerung verzichtet werden. Die Orchestrierung des Ladevorgangs im ODI wurde mit den ODI-Elementen „Globale Variablen", „Packages", „Procedures" und „Load Plans" realisiert. Packages erwiesen sich als nützliches Strukturierungselement, um zusammengehörige Abläufe transparent zu kapseln. Dabei konnte eine sinnvoll parallelisierte Anordnung von Prozessen zu einer insgesamt deutlich optimierten Auslastung der Rechner-Ressourcen und zu reduzierten Ladezeiten beitragen *(siehe Abbildung 6)*.

Apex zum Reporting und zur Metadaten-Pflege

Zur Vereinheitlichung der zum Teil semantisch unterschiedlich interpretierbaren Daten der Vorsysteme war eine Bereitstellung von Lookup-Tabellen erforderlich, um eine vereinheitlichte Darstellung im CORE ablegen zu können. Die Pflege der fachlichen Codes und deren vereinheitlichte Übersetzung wurden mithilfe einer Apex-Anwendung realisiert und dem verantwortlichen Fachbereich zu Verfügung gestellt.

Um einen geregelten Zugriffschutz zu erreichen, wurde zur Authentifizierung und Autorisierung in Apex der LDAP-Service des Unternehmens angebunden. Durch Einsatz von Oracle Virtual Private Database ist eine hohe integrierte Datensicherheit für die unterschiedlichen Zugriffsgruppen aus den Fachbereichen sichergestellt.

Als Interimswerkzeug vor Verfügbarkeit des BI-Tools MicroStrategy wurde Apex sogar zur Erstellung von KPI-Reports eingesetzt. Es waren dabei mit vertretbarem Aufwand Berichte realisierbar, die fest vordefinierte Drill Downs ermöglichen.

Aufgetretene Probleme

Konnte der Einsatz neuester Oracle-Technologien zu einem reibungslosen Projektfortschritt beitragen? Die wesentlichen Projektziele wurden im Rahmen des selbst gesteckten Zeitrahmens trotz einiger Schwierigkeiten erreicht. Der eine oder andere Service Request musste gestellt werden und so waren auch neue Versions-Einspielungen in Datenbank und ODI erforderlich, um die Probleme zu beheben.

Es ist positiv hervorzuheben, dass es mit dem ODI keine projektkritischen Komplikationen gab. Im Vergleich zu den OWB-Versionssprüngen aus der Vergangenheit hat sich ODI 12*c* in Anbetracht der teils vielfältigen Erweiterungen gegenüber der 11er-Version gut bewähren können – und hat die Erwartungen übertroffen.

Fazit

Wie sind die im Projekt gemachten Erfahrungen im Vergleich zum OWB einzuordnen? In Summe wurde der OWB im Rahmen des Projekts nicht wirklich vermisst. Im Gegenteil, der ODI-komplexe Mechanismus, wie SCD 2, kann elegant in den Knowledge-Modulen kapseln und hilft damit, die Entwicklungszeit zu reduzieren. Eine Bewertung im Einzelnen:

- *Komplexe Mappings mit dem ODI? Geht das?*
  Dem ODI ist in 12*c* aufgrund neu spendierter Component-Style-Knowledge-Module die Fähigkeit gegeben worden, Mappings ähnlich zu strukturieren, wie es im OWB möglich war. Das ist wie im OWB eine Voraussetzung für vollständige Lineage- und Impact-Analysen. Außerdem erzeugt der ODI SQL-Kommandos und keine PL/SQL-Packages wie der OWB. Die Darstellung einzelner SQL-Steps im Runtime Repository erlaubt dabei einfacheres Monitoring.

- *Automatisierte Mapping-Generierung mit Groovy*
  Der Einsatz der Groovy Scripting Language ermöglichte eine automatisierte Generierung von Mappings, wie es im OWB mit der Tcl-Sprache OMB*Plus möglich ist. Das an Java erinnernde Groovy überraschte mit einfacherer Handhabbarkeit als die gewöhnungsbedürftige TCL-Syntax und so konnten alle Erwartungen erfüllt und teilweise auch übertroffen werden.

- *Einsatz von Load Plans*
  Mit den Elementen „Variablen", „Procedures" und „Packages" lassen sich komplexe Load Plans implementieren. Parallele Prozess-Verarbeitungen sind möglich und so blieben im Projekt keine Wünsche im Vergleich zur Workflow-Technik des OWB offen. Gelegentliche Repository-Korruptionen beim Speichern von Load Plans trübten das Bild allerdings ein wenig.

- *Performance mit dem ODI*
  Ein direkter Vergleich ist hier schwer zu treffen. Beide Tools generieren Code, der auf der Ziel-Datenbank ausgeführt wird. Die KMs haben den Vorteil, dass der generierte Code direkt beeinflussbar ist. So kann dieser bei Bedarf angepasst und so bei Performance-Problemen zielgerichtet reagiert werden. Zu erwähnen sei hier der Umstand, dass die neu eingeführten Component-Style-KMs nicht anpassbar sind. Man darf gespannt sein, ob zukünftige ODI-Versionen hierzu Konfigurationsmöglichkeiten eröffnen.

- *Historisierung out of the box?*
  Wie schnell eine SCD2-Historisierung mit KM realisierbar ist, hat positiv überrascht. Diese Technologie hat großes Potenzial, um Komplexität zu reduzieren und damit die Produktivität bei der ETL-Entwicklung zu steigern. Dass die SCD2-Terminierung von gelöschten Quelldatensätzen out of the box nicht als Bestandteil des KM mitgeliefert wird, konnte man durch Erweiterung des KM wieder kompensieren.

- *Metadaten-Repository*
  Die Laufzeit-Informationen des ODI-Repository, die über den Operator „Tabulator" abgefragt werden können, liefern ausreichend Informationen, um Fehler rasch zu identifizieren und zu beheben. Im Gegensatz zum OWB war es im Projekt nicht erforderlich, zusätzliche „Self-made"-Repository-Queries bereitzustellen. Gelegentlich auftretende Repository-Korruptionen, deren Ursache vor der Verfügbarkeit geeigneter Patches nicht geklärt waren, veranlassten das Team, einen Metadaten-Mart aufzubauen, um Runtime-Informationen zu persistieren.

Quellenverzeichnis

[1] ODI 12*c* - Mapping SDK the ins and outs: https://blogs.oracle.com/dataintegration/entry/odi_12c_mapping_sdk_the

[2] Maßgeschneidertes & performantes ETL durch ODI-Knowledge-Modul-Modifikationen: https://blog.mt-ag.com/index.php/2016/02/11/odi-km-modifikationen

Bernhard Rosenberger
bernhard.rosenberger@mt-ag.com