# 1 MDA und Tools

Die Entwicklergemeinde kennt ein neues Zauberwort, das in traditioneller Form der drei Buchstaben daherkommt: MDA (Model Driven Architecture). Wird man nun weniger programmieren und vermehrt modellieren, oder induziert die Toolindustrie wieder neue Bedürfnisse ? Beides, den eine stärkere Formalisierung ist sowohl der Wunsch von uns geplagten Entwicklern wie auch die Bedingung der Industrie, Tools und Architekturen zu vereinheitlichen. All diese Vorhaben in UML 2.0 laufen unter dem Begriff MDA zusammen. Das Ziel ist es, ausführbare Modelle zu generieren.

## 1.1 MDA im Banne der Technik

In einem ersten Teil möchte ich die Grundzüge der MDA aufzeigen, der zweite Teil ist dem praktischen Einsatz der MDA inklusive Projektstudie gewidmet.Der dritte Teil stellt die zugehörigen Tools vor. Solche CASE-Tools, die MDA als Etikett andrucken, erleben momentan ein Aufblühen, bei näherer Betrachtung aber eher ein Verglühen.

John Siegel, Vice President of Technology Transfer bei der Object Management Group (OMG), spricht von einer "enthusiastischen" Reaktion des Marktes auf MDA. Nach Siegel wird MDA schneller angenommen als jeder OMG-Standard vorher – und dazu gehören immerhin so weit verbreitete Standards wie Corba oder UML.

Ich kann dem hinzufügen, dass man den Begriff MDA fast inflationär braucht. Sein oder Design ist die Gunst der Stunde. Neue Geschäftszweige bauen sich auf, wie das EDOC (Enterprise Distributed Object Computing) oder die PDA (Process Driven Architecture). Das erklärte Ziel der OMG in Bezug auf EDOC ist bspw. die Vereinfachung der Komponentenentwicklung mit Hilfe eines auf UML 1.4 basierenden Frameworks, das zusätzlich noch mit der MDA kompatibel ist. Bei allem Respekt, aber die technologieunabhängigen Metamodelle stehen hier noch ziemlich am Anfang in Bezug auf EDOC. Jedes Metamodell der UML ist eben so gut wie seine Umsetzung, UML 2.0 hat noch einige Arbeit zu bewältigen.

Die letzte freigegebene Version von UML betrifft den Release 1.4, den die OMG im Mai 2001 verabschiedete. Der Release 1.4 betraf vor allem Erweiterungen in der Geschäftsprozessmodellierung sowie Anpassungen an die OCL (siehe Glossar).

Seit dieser Zeit arbeitet man an der Version 2.0, die den modellgetriebenen Ansatz (MDA) in den Vordergrund stellen will. Patterns bilden darin die Brücke zwischen der fachlichen und der technischen Sicht.[i] Ivar Jacobson erwähnte an einem Seminar im Dezember 02 in Zürich den Begriff „Executable UML“, welcher angeblich einer der fünf Makrotrends im künftigen Softwarebau sein soll.

Daß ein Architekturstil nicht nur etwas mit dem Bau und dem Zusammenspiel von Komponenten zwischen den Schichten zu tun hat, sondern auch mit den Personen und Rollen in der Organisation, die solche Komponenten generieren, macht MDA als Technik wie auch als Prozeß interessant. Diese Einsicht kam schon im Juni 99 zu Tage, als das Gremium der Object Management Group (OMG) [[ii]] eine neuartige Codegenerierung aus Modellen und Anforderungen begründete.

Wenn aber die Anforderungen schon im Modellansatz bestehen, spricht wenig dagegen, diese modellgetriebene Architektur bei ähnlichen Anforderungen immer wieder einzusetzen. MDA gibt es bereits für existierende Software, kommerziell wie auch OpenSource[[iii]], die sich mittels Reverse Engineering umwandeln läßt. Zumal MDA auf der höheren Abstraktion ja sprachunabhängig ist. Nun, was meint Model Driven Architecture im Alltag?

Jede Sprache mit der zugehörigen Entwicklungsumgebung ist von einer inneren Architektur von Subsystemen abhängig, z.B. dem Framework der GUI, der Trennung von Interface und Implementierung oder den typischen Datenbankzugriffs-Komponenten mit ihren Connection-Strings.

> *Ein MDA-Generator sollte diese Architektur kennen, damit man sich in den Modellen nicht um diese technischen Details kümmern muß.*

Mein Paketdiagramm sollte also bereits wissen, ob ich z.B. dbExpress mit SOAP statt ADO mit COM einsetzen werde, wenn ich das Modell zum Generieren einsetzen will!

Um diese „Probleme“ bei der Generierung zu lösen, schlägt die OMG vor, ein „Platform Independent Model“ (PIM) durch einen MDA-Generator zu erzeugen. Ein PIM ist ein UML-Modell, in dem die technischen Details nicht ersichtlich sind. Somit befindet sich das Modell auf einer höheren Abstraktionsebene und ist unabhängig von technischen Fakten wie der bekannten Programmiersprache oder Datenbanktechnik.

Das PIM muß aber genug Details besitzen, um das generische Modell durch definierte Abbildungsregeln und Architekturmuster mit einem plattformspezifisches Modell (PSM) auf die Zielsprache abzubilden, d.h. innerhalb der Plattform (siehe Glossar) mit zu generieren.

Im Klartext heisst dies, es lässt sich nichts generieren, was nicht schon vorher in Form einer Referenzimplementation validiert, plausibilisiert und bestätigt wurde! Diese Referenz wird dann in Form von konkreten UML-Profilen weitergegeben. Skalierbarkeit, Wartbarkeit und Perfomance sind dann genauso gut wie die Referenz, aus der man die Abbildungsregeln ableitet (auch Transformation genannt).

Zurzeit existieren aber noch keine OMG-Spezifikationen für eine standardisierte Transformationssprache. Also, wo MDA draufsteht muss es nicht drin sein. Angenommen es gibt ein UML-Profil für dbExpress/CLX und eines für Ado.NET oder EJB (EnterpriseJavaBean). Mit einer Standardisierung der Transformation liessen sich dann die Profile austauschen!

> *Ein PSM (Platform Specific Model) ist das spezifische und technisch abhängige Model, das die Architektur erzeugt.*

Wer sich jetzt fragt, wie die Abgrenzung zu Design Patterns zu setzen ist, der weiß aus eigener Erfahrung, daß ein Tool aus einem Patternkatalog direkt sprachspezifischen Code produziert. So etwas wie einen Design-Pattern-Generator für Geschäftsprozesse gibt es noch nicht, aber einige Tools sind in dieser Hinsicht schon weit fortgeschritten.

> *Also positioniert sich die MDA einen Schritt vor dem Design und ermöglicht bereits in der Analyse sprachunabhängige Klassendiagramme zu generieren.*

MDA ist also mehr als ein Pattern-Expander, denn beim Erzeugen eines Pattern und den daraus resultierenden Klassenrümpfen ist dieser Schritt einmalig und nicht umkehrbar. MDA lässt sich aber wiederholt unter Erhaltung des eigenen Codes (geschützter Code) generativ einsetzen. Wobei auch die MDA nicht umkehrbar ist, will heissen aus einem generieten PSM lässt sich kein PIM mehr gewinnen.

## 1.2 Kernidee

Kernidee der MDA ist also die schrittweise Verfeinerung von der Analyse zum Design ausgehend von einer Modellierung der fachlichen Applikations-/Geschäftslogik, die völlig unabhängig von der technischen Implementierung und der umgebenden IT-Infrastruktur ist. Das Grundprinzip zu MDA lässt sich wie folgt darstellen:

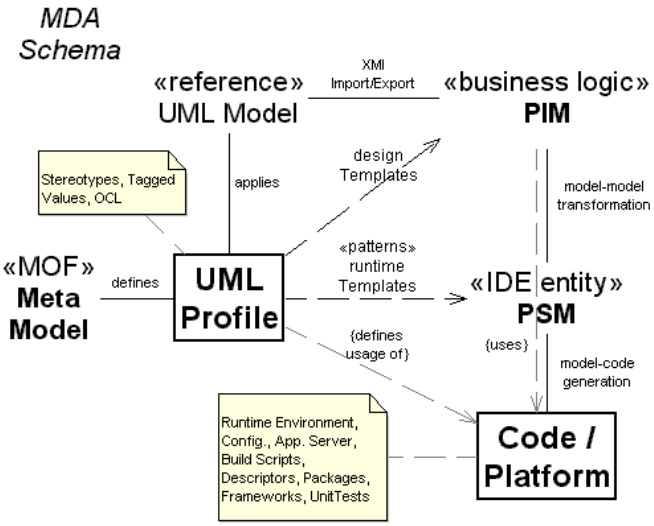


Abb. 1.1: Das MDA Prinzipschema

Das Schema soll den Weg vom abstrakten Metamodell über die unabhängige Fachlogik zum konkreten Code auf der Plattform verdeutlichen, ähnlich einer CORBA-Spezifikation, der IDL (Interface Definition Language) und dem daraus folgenden Code. Was bedeuted dies nun in der Praxis?

Man beginnt mit der Erstellung des Designs oder einer Referenz mit einem konventionellen UML-Tool. Das Tool muss aber fähig sein, mit UML-Profilen und constraints arbeiten zu können. Durch einen XMI-Export erhält man eine austauschbare Designdatei (PIM), die man mit dem gewünschten UML-Profil einem MDA-Generator füttert. Der Generator transformiert die Designdatei mit dem Profil nach spezifizierten Abbildungsregeln (Templates) in einen konkreten Architekturrahmen als typisierte Packages. Nun hat man das PSM, das immer noch ein Modell ist.

Mit der Generierung wird dann erstmals Code im Sinne eines Implementationsrahmen erzeugt. Die geschützten Bereiche sind zunächst noch leer. Es erfolgt dann die Entwicklung der eigentlichen Fachlogik in den Klassen, die mit einer weiteren Generierung durch Interpreter oder Compiler zum ausführbaren Code bezüglich der gewählten Plattform führt. Die einzelnen Schritte:

- Modelliere das PIM
- Transformieren auf ein PSM
- Generieren von Code aus dem PSM
- Codieren der Fachlogik
- Integrieren in die Plattform

Wenn nun das Tool aus dem PIM ein plattformspezifisches Modell (PSM) generiert, sollte Einigkeit über die zu implementierende Plattform herrschen. Ist nun eine Plattform CLX, .NET oder J2EE, dann sollten die entsprechenden Klassen und Typen zum Zuge kommen, wie J2EE 1.2 compliant oder CORBA 2.3 compliant ORB. Was aber ist, wenn die Plattform eine Makro-Sprache mit Objektmodell oder sogar eine Spezifikation wie CORBA ohne Applikationsserver hat?

> *Der PSM UML-Generator muss also nicht nur die Zielsprache, sondern auch die Zielplattform aus den Profilen kennen.*

Im Hinblick auf die Ausarbeitung solcher Details stehen die MDA-Bewegung und die Toolhersteller noch ziemlich am Anfang. Zusätzlich soll in die MDA das Wissen über die zugehörige Businessdomäne einfliessen.

Auch das Wissen über die zugehörige Fachlogik soll einfließen. Für Domänen wie z.B. Finanz- oder Versicherungswesen, Telekommunikation, Medizinaltechnik sollten die typischen abstrakten Prozesse schon vordefiniert sein und für die Erstellung der PIM als sogenannte „domain-specific core models“

bereitgestellt werden. Ein erster Ansatz ist Statemate, ausführbare UML-Diagramme für Embedded Systeme.[iv] Für den Austausch der Modellinformationen über Toolgrenzen hinweg wird XML Metadata Interchange (XMI) eingesetzt. Der Name zeigt es bereits deutlich: XMI ist ein Mitglied der XML-Sprachfamilie. Die XMI-Norm definiert für das UML-Metamodell eine XML Document Type Definition (DTD) oder ein Schema. Sie stellt die Grammatik der Sprache zur Verfügung. XMI ist ebenfalls ein Standard der OMG.

Bei der Transformation von einem abstrakten Modell hin zu einem konkreteren Modell soll nicht nur Information über die technische Infrastruktur verwendet werden, sondern es wird auch Wissen berücksichtigt über die Businessdomäne, in der die Applikation eingesetzt werden soll (Praxisbezug).

Der MDA-Ansatz birgt zusätzliches Potential für die Wiederverwendbarkeit von Modellen, da diese ja plattformunabhängig sind. Durch den Einsatz von standardisierter Transformation ist auch die Qualitätssicherung erhöht. Nach OMG sind dies die Hauptvorteile von MDA:

- Reduzierte Kosten bei der Entwicklung
- Erhöhte Qualitätssicherung durch Selbstähnlichkeit
- Schnellere Integration neuer Technologien

Als Nachteil von MDA muss man erwähnen, dass strikte nach Forward Engineering gearbeitet wird. Ein Beispiel ist das Umbenennen von Klassen und Methoden. Nimmt man eine solche Änderung am Code vor oder ändert man das Modell entsprechend, aktualisiert das Tool die jeweils andere Seite automatisch. Typische Refactoring Techniken, die auch in einem Review zum Tragen kommen, sind bei MDA ein Problem.

Am 6. Januar 2003 wurde die überarbeitete Fassung der OMG übergeben. Am Ende dieses Jahres, da erfahrungsgemäß die Finalization Task Force 9 Monate dauert, erwartet die Entwicklergemeinde eine Freigabe der Version 2. Diese teilt sich auf in drei separate aber zusammenhängende Gebiete:

1. **UML Infrastructure**, welche eine Vereinfachung und Modularisierung des Metamodells vorsieht und keinen Einfluß auf den Benutzer hat. Dieses Gebiet ist für die Toolhersteller und Methodiker interessant, die an Profilen und möglichen Stereotypen Freude und Nutzen haben. Zudem wird das Metamodell von sprachabhängigen Konstrukten (wie C++ oder ADA) gesäubert und sprachneutral, d.h. in OCL definiert.
2. **UML Superstructure**, mit den eigentlichen Erweiterungen bezüglich der Notation und Syntax und somit für den Benutzer sichtbar und verwertbar. Ziemlich aufgewerten will die OMG die komponentenbasierte Entwicklung mit z.B. einer präziseren Definition einer Schnittstelle mit erweiterten Signalen oder Nachrichten. Einige statische und dynamische Elemente werden auf Echtzeitfähigkeit getrimmt.
3. **UML OCL**, die eine erhöhte Integration in das Metamodell und die Syntax vorsieht, so daß die Spezifikation durch ein Modell zum Code auch eine formale Sprache beinhaltet. Die OCL (siehe Glossar) wird von einer Sprache der Bedingungen zu einer generellen Ausdruckssprache erweitert. Weiter folgt der verbesserte Modellaustausch durch die XMI (XML Metadata Interchange) z.B. mit Layout Informationen zwischen den einzelnen Tools.

Es ist klar, das auf der einen Seite die Toolhersteller nun gefordert sind. Auf der anderen Seite wird die Vision des „Executable UML“ eine gravierende Änderung des Entwickleralltages zur Folge haben, sofern die Welt und schlußendlich auch unsere Kunden mit „standardisierter Individualsoftware“ und einheitlichen Geschäftsprozessen zufrieden sind. Etwas in dieser Art ist beim BizTalk Server von MS zu finden, der eine Generierung des Dokumenten-Workflows erlaubt.

Mehr noch, bezüglich der Wartung und Pflege einer Anwendung sind dann die selben Fehler zu erwarten, da aus dem weltweiten Modellkatalog in den Codegeneratoren auch ähnliche Fehler entstehen können.

Die wichtigsten Neuerungen (über 540 wurden im Sommer 02 der Revision Task Force überreicht) des Release 2 folgen nun in geraffter Form auf einen Blick:

- Für das Metamodell entsteht ein Sprachkernel

- Zusätzliche Notationen bauen auf dem Kernel auf
- Support für den Einsatz von Komponenten untereinander
- Ausbau der Geschäftsprozessmodellierung (BPM)
- Interne Struktur für Bezeichner, Schnittstellen, Klassen, Typen und Rollen
- OCL Integration als generelle Spezifikationssprache eines Modells
- State Event Generalisierungen und Echtzeiterweiterungen
- Erweiterungen für kompaktere und wiederverwendbare Sequenzdiagramme
- Präziseres Abbilden der Notation zur Syntax

## 1.3 Praxis

Die UML bietet grafische Ausdrucksmittel an, um Anforderungen zu definieren sowie ein System fachlich und technisch zu entwerfen. Allerdings macht sie keine Aussage, wann, wie und von wem diese Mittel im Projektverlauf zum Zuge kommen.

Nun, wie man zum Modell/Code gelangt ist eher eine Frage der Technik, z.B. mit Design Patterns, oder man generiert mit der MDA (Model Driven Architecture) schon in einer frühen Phase ganze Architekturpatterns [V]. Auch wenn es mittlerweile standardisierte Idiome wie getter und setter-Methoden, IDL-Generierung oder Listenklassen gibt, die sich mit Code-Templates auch erzeugen lassen, hört die eigentliche Generierung innerhalb der Methode einer Klasse auf. Die Geschäftslogik eines Systems, d.h. ihr eigentlicher Wert, wird auf einem hohen Abstraktionsniveau in einem plattformunabhängigen Modell modelliert.

In der Praxis sind bereits fertige PIM's erhältlich. Über automatisierte Transformationen in Modelle niederen Abstraktionsgrades werden dem PIM implementierungsspezifische Details hinzugefügt, wobei sogenannte plattformspezifische Modelle entstehen. Ein PSM dient dann als Grundlage für die Generierung aller Infrastrukturkomponenten der Anwendung (Code, Configuration, Test- und Build-Umgebung, Deployment-Infrastruktur). Ein PSM lässt sich dann konkret als Package- oder Component Diagramm generieren.

Auf der anderen Seite gibt es Prozesse, die ein modellgesteuertes Vorgehen vorsehen, so dass z.B. aus dem Paketdiagramm auf Stufe Design Patterns möglich ist, Code fast konstruktionsfertig zu erzeugen. In der Tabelle 1.2 zeige ich, basierend auf dem MDA Schema, eine Übersicht zu den einzelnen Schritten. Fast allen Schritten gemeinsam ist der Einsatz der UML-Notation und Objektorientierung (OO), iteratives Vorgehen sowie ein vermehrtes Ausrichten auf Patterns. Auch die MDA ist eine Art prozessorientiertes Architekturmuster:

| Schritt | Techniken | Beispiel | Output | Transformer |
|---|---|---|---|---|
| Fach Spezifikation Analyse /Design | UML-Profile OCL | CD-Album Verwaltung | PIM | Model-Model Transformation |
| Technisches Modell Design | UML-Profile Patterns | XML-File Storing | PSM | Model-Code Transformation |
| Implementierung | Compiler | Grid&File | Code | Micromethode |
| Integration | Compiler | .NET | Plattform | Macromethode |

Tab. 1.2: Prozessmodelle im Vergleich.

Codieren wie Architekten, das klingt gut. Wichtig ist hier, bereits am Anfang eine klare Trennung von fachlichen und technischen Anteilen in den Klassen zu erreichen. In der Praxis kann man hier bereits einen Hauch von MDA in sein Klassendesign einfliessen lassen. Stellen Sie sich eine Klasse TCustomer vor, mit den Methoden Store und Undo. Store und Undo sind technische Aspekte, die mit der Fachwelt des Kunden nichts zu tun haben. Oder kennen Sie einen Kunden, der als Mensch ein Undo besitzt. Ich kann ja auch nicht meinen vor Jahren erlebten Unfall rückgängig machen!

Im PSM sind in der Praxis Informationen über die eingesetzte Softwareinfrastruktur enthalten, wie z.B. ein J2EE-Applikationsserver, ein dbExpress Provider, oder ein ADO.NET-Zugriff.

Die Transformation kann dann ausgehend vom PIM über mehrere unterschiedlich detaillierte PSM hin bis zum Code erfolgen. In der MDA sind die notwendigen Schritte bei dieser Transformation definiert und durch den Einsatz von Tools halb oder vollständig automatisierbar. Diese Techniken generativer Softwareentwicklung sind mit MDA-Konzepten und praxiserprobten Architekturbausteinen möglich.

Der Vorteil: Beim Austausch einer Middleware muß der Entwickler also nur die geeignete regelbasierte Transformation verwenden, um aus dem unveränderten PIM ein neues, passendes PSM zu generieren.

Zentral ist hier das Modellieren der Geschäftslogik und -anwendung ohne Details zu ihrer technischen Umsetzung. Diese Trennung ist am Anfang recht gewöhnungsbedürftig, da man oft an irgendwelche technische Methoden aus dem Prototyping gebunden ist. Darauf aufbauend werden technologiespezifische Modelle erstellt, die eine konkrete Umsetzung in einer gewählten Technologie, wie z.B. J2EE, CLX, oder .NET beschreiben.

> *Die Aufteilung in plattformunabhängige und plattformspezifische Modelle ermöglicht eine langfristige Anpassbarkeit an zukünftige Technologien, ähnlich dem Auswechseln des Datenbank-Providers.*

Auch in der modernen Architektur ist der Detaillierungsgrad anscheinend verlorengegangen: Für auf dem Reissbrett entstandene Architektur haben sich Ornamente und Schnörkel an der Aussenfassade als unnötig erwiesen. Doch die innere Komplexität der Gebäude ist gewachsen. Aussenwände sogenannter intelligenter Gebäude (Intelligent Buildings) verbergen hinter der glatten Fassade ein äußerst kompliziertes Innenleben.

Damit wird klar, daß nebst der Architektur auch die Funktionalität einen hohen Detaillierungsgrad aufweisen kann.

> *Der Trend im Softwarebau wird sein, die Architektur zu standardisieren (J2EE, CLX, .NET) aber die Funktionalität zu erhöhen.*

Standardisieren kann auch vereinfachen. Denn in den seltensten Fällen wird künftig jeder Teil eines Modells ein anderer Sprachentyp sein, das entspräche der Arbeit mit Einzelelementen.

## 1.4 MDA Projekt

In dieser Kurzübersicht will ich den Einsatz von MDA nun pragmatisch darstellen. Ich werde die in der Tabelle erwähnten vier Schritte von MDA durchgängig erklären. Beim Projekt handelt es sich um eine Musikverwaltung mit den entsprechenden Medien wie Audio-CD oder Schallplatten. Als Tool kommt ModelMaker und ein selbstgebauter pseudo MDA/D-Generator (D für Development) mit Templatefunktion zum Einsatz. Dieser Generator hat aber nur didaktischen Nutzen, demzufolge soll die Grundidee von MDA praktisch nachvollziehbar sein. Eine erste Umsetzung des Pseudogenerators war die Weiterentwicklung von DelphiWebStart. [vi] Ich habe das Projekt mittlerweile der Community zugänglich gemacht und hoffe auf rege Mitarbeit unter:

http://sourceforge.net/projects/delphiwebstart

Wie oft hatten Sie schon die Idee, einem bestimmten Klassendesign aus fachlicher Sicht auch gleich die Bauanweisung für Speicherung und Darstellung der Daten mitzugeben. Denn die Umgebung von Fachanwendungen impliziert meistens eine ähnliche Struktur, so daß weitere Anwendungen wie eine Instanz aus einem Metamodell oder dem konkreten Profil entstehen könnten. Wenn man dann ein Grid (Gitter) generieren lässt möchte man auch gleich das Navigieren zwischen dem Grid und der Einzelansicht als Maske miterledigt haben.

Mit einem ersten PIM und den Fachklassen `TMusicRecord`, `TArtist` und `TTrack` ist die Trennung der technischen Aspekte vollzogen. Ein UseCase in der Anforderungsanalyse führt indirekt zum vorliedenden PIM, das keine sprachspezifischen Typen aufweist.

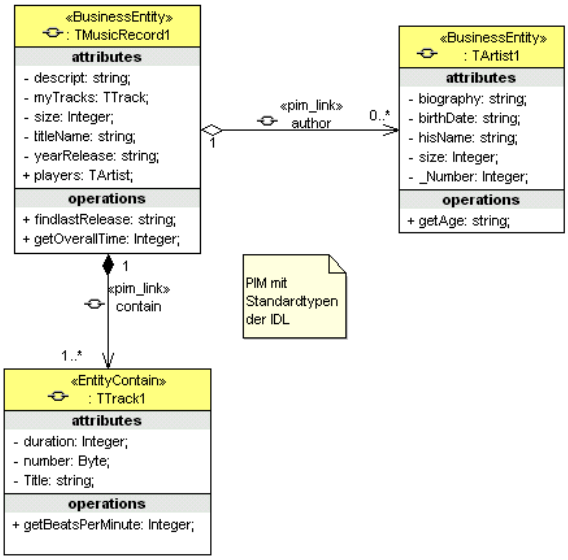


Abb. 1.3: Unser erstes PIM als XMI Import

Dieser PIM soll jetzt mit einem Profile ein weiteres Modell erzeugen, das mir die Probleme der Persistenz und der Darstellung lösen soll. Mir schwebt vor, die Daten der Musikmedien filebasiert zu speichern und in einem Grid bearbeiten zu können. Die Geschäftsprozesse wie Suchen von Artisten oder Erstellen einer Statistik habe ich vorgängig in einem Activity-Diagramm modelliert. Zwischen einem Activity und der MDA gibt es keinen direkten Zusammenhang, die MDA geht bei einem PIM von Klassen- oder Zustandsdiagrammen aus, die Implementierung der Fachlogik erfolgt nach herkömmlicher UML-Art.

Was ich nun benötige ist das vorgängig erstellte Profil, das aus einer manuell erstellten Referenzimplementierung entstanden ist. Es ist kompliziert und auch wenig sinnvoll, direkt mit einer Template-Entwicklung zu beginnnen, wenn noch keine Basis-Architektur vorliegt. So habe ich die Mechanik der Speicherung und Darstellung zuerst gebaut und dann mit Tags und Makros sozusagen ein Template als Profil erstellt. Dieses Profil will ich künftig DStoreGrid nennen und als Template speichern:

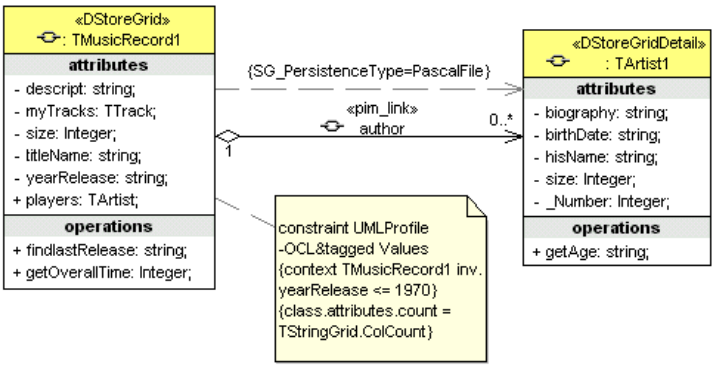


Abb. 1.4: Das Profil ist aus einer Referenz entstanden

Zu sehen ist bspw. eine Constraint die besagt, dass die Anzahl Kolonnen im StringGrid genau der Anzahl Attribute der Klassen entspricht, bezogen auf die einfachen Datentypen (descript, size, titleName und yearRelease).

```
{class.attributes.count = TStringGrid.ColCount}
```

Die beiden Typen `TTrack` und `TArtist` sind Master-Detail Beziehungen die mit Listen oder Kollektionen aufgelöst werden. Die Klasse `TTrack` ist im Profil nicht ersichtlich, gehört aber im Template mit dazu. Auch eine Tagged Value ist mit von der Partie, das Profil kennt nun die Art der Speicherung als PascalFile, die als constraint modelliert ist. Ein weiteres Value {yearRelease <= 1975} macht auf einen Schlag das gesamte System zu einer Oldies-Sammlung ;).

Mit dem Erweitern durch die Profile in UML 2.0, welches eine Art Wörterbuch von Stereotypen und Values darstellt, erhalten auch Packages eine Aufwertung.

> *Auf diese Weise erhalten Klassen oder Pakete innerhalb einer gemeinsamen Fachdomäne ihr eigenes Stereotypenpaket, z.B. das Profil Musikmarkt.*

Ein Profil mit den zugehörigen Typen und sogar Operationen wie `getOverallTime` erlaubt, in einem Durchzug und in allen Diagrammen den Typ Integer auf Float oder DateTime zu wechseln! Ein Profil sollte gemäß der Spezifikation von Version 2 auch Toolübergreifend austauschbar und einsetzbar sein.

Das Klassendiagramm wurde in UML 2.0 verfeinert und im Hinblick auf den vermehrten Einsatz von Komponenten erweitert. Alle bestehenden Notationen wurden unverändert übernommen. Klassen und Interfaces erhalten neu einen <part>, das ist ein Teil einer Klasse. Dieses Teil (im wahrsten Sinn des Wortes) ermöglicht in einer frühen Phase den Typ einer Klasse zu bestimmen, der dann später in der Implementierung mit Operationen und Attributen ausgearbeitet wird.

Was nun folgt ist das transformierte PSM mit den zugehörigen technischen Details:

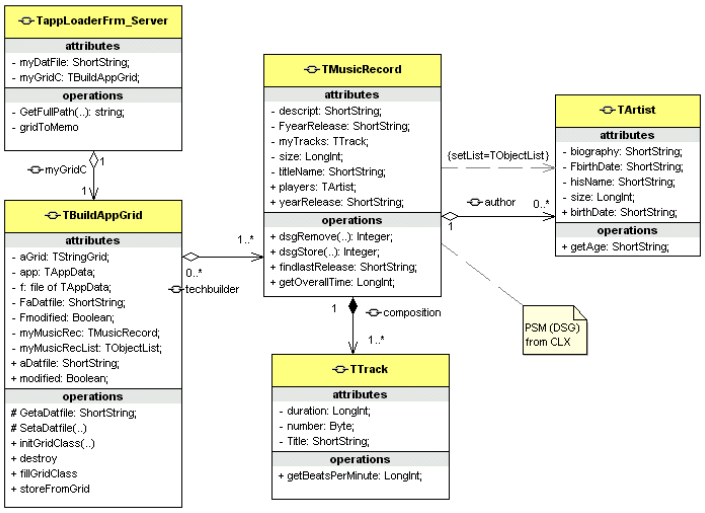


Abb. 1.5: Aus der Transformation entstanden, das PSM

Diese PSM erhält den Namensraum DSG (DelphiStoreGrid) und wird als Package gespeichert. Auch die Fachklassen sind nun mit technischen Details wie dsgRemove oder dsgStore angereichert, Design Patterns wie der Builder, Composite oder die Facade sind im definierten Package mit eingebaut.

Packages sind in UML 2.0 genauer geworden, das Anzeigen der Klassen erhöht den Informationsgehalt, zudem läßt sich nun zwischen <access> und <import> als Abhängigkeitslinie unterscheiden; bei <access> greift eine Klasse auf die öffentlichen Elemente der anderen Klasse zu, bei <import> wird der ganze Geltungsbereich eines Paketes dem anderen Paket hinzugefügt oder eben

importiert, z.B. bei einer IDL, DLL oder bei Type Libraries. Packages sind nach der Spezifikation der UML ein Mechanismus zur Gruppierung von Design Objekten.

Auch die Komponenten sind in UML 2.0 aufgewertet. Vermehrt ermöglichen <ports> und <connectors> zwischen einer angebotenen (Technik, Typ, Laufzeit und Signatur) und einer erforderlichen Schnittstelle das mögliche „Zusammenstecken“ prüfen zu lassen, um z.B. festzustellen, daß eine MSCOM nicht zu einem EJB oder einer CLX kompatibel ist. Durch die Detailsicht mit Hilfe von <ports> sind neu, geschachtelte Komponenten oder komplizierte Strukturen wie ganze Container besser beschreibbar.

Aus diesem PSM lässt sich nun im letzten Schritt der sogenannte Implementationsrahmen erzeugen, der dann mit Fachlogik, wie die Methode findLastRelease, weiter ausgebaut wird. Diese Bereiche sind dann als gschützt markiert, so dass bei der nächsten Generierung kein Überschreiben erfolgt. Werfen wir noch einen Blick in den Code:

```
{***************** TBuildAppGrid *****************}
constructor TBuildAppGrid.initGridClass(vGrid: TStringGrid;
                                vFile: shortString);
begin
  aGrid:= vGrid;
  aDatfile:= vFile;
  myMusicRecList:= TObjectList.create;
  myMusicRecList.ownsObjects:= true;
  with aGrid do begin
    ScrollBars:= ssAutoVertical;
    FixedRows := 1;
    FixedCols:= 0;
    ColCount:= 4;
    RowCount:= 2; //title is one row
  end;
end;
```

Der von der PSM generierte und wiederverwendbare Code hat vor allem die technischen Details gelöst, auszugsweise der Abschnitt wo das Einlesen der Daten aus dem Filesystem in die Objekte erfolgt und die Implementierung zusätzlich das im Constructor übergebene StringGrid mit den Musikdaten abfüllt:

Bsp: [Led Zeppelin II, 38:90, 1972, Sec. Album by Zep with Whole Lotta Love Single]

```
.....
     AssignFile(f,aDatFile);
     Reset(f);
      try
        while not Eof(F) do begin
          Read (F, app);
          myMusicRec:= TMusicRecord.create;
          with myMusicRec do begin
            titleName:= app.Name;
            size:= app.Size;
            FyearRelease:= app.Release;
            descript:=  app.descript;
            Cells[0,crow]:= titleName;
            Cells[1,crow]:= intToStr(size);
            Cells[2,crow]:= FyearRelease;
            Cells[3,crow]:= descript;
            myMusicRecList.add(myMusicRec);
```

```
          end;
          Inc(cRow);
          RowCount:= cRow +1;  //new entry
        end;
      finally
.....
```

In einem weiteren Ausbau ist sicher ein Profil zur Datenbankspeicherung gefragt, das in einem generativen Sinne im ersten Entwurf wie folgt aussehen könnte:

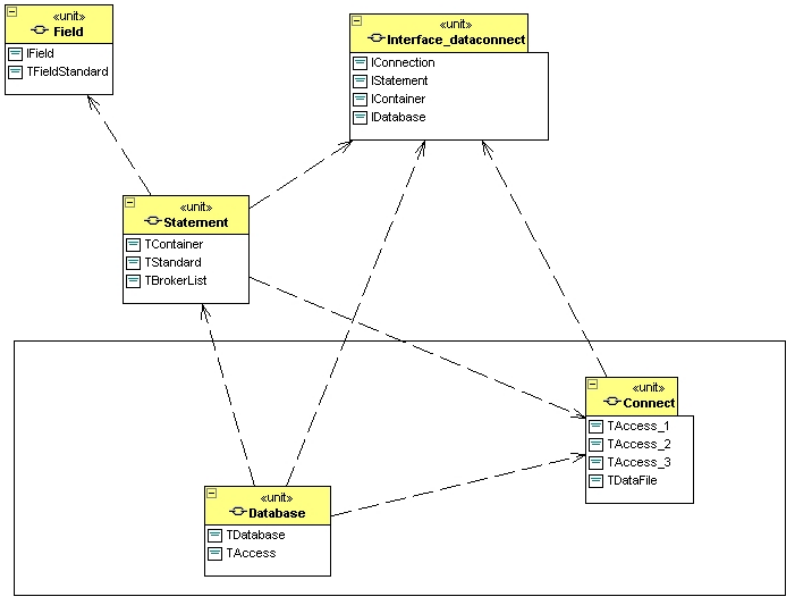


Abb. 1.6: Start zum Bau eines neuen Profiles „dbExpress/CLX“

Weitere Techniken sind im Gespräch, die alle eine gewisse Gemeinsamkeit mit MDA aufweisen, die sich „generative Programmierung mit adaptiven Mustern“ nennt. Diese Technik lohnt sich vor allem dort, wo man lange in der gleichen Fachrichtung entwickelt, also gute Kenntnisse der „Business Domain“ hat.

Bei der Entwicklung eines CAD-Tools oder einer Software aus der Regelungstechnik sehe ich kaum eine Chance, von einer besseren Wiederverwendbarkeit als bisher zu profitieren. Auch der Begriff „Intentional Programming“[vii] gehört zur Thematik, welcher vorderhand ein interessanter MS-Traum ist, aber durchaus Potential im Zusammenhang mit .NET besitzt.

Muster im Großen wie im Kleinen, von Architekturmustern über Entwurfsmuster bis hin zu Codemustern, spielen bei der objektorientierten Entwicklung eine Rolle, wenn es darum geht, Effizienz und Qualität gleichermaßen entscheidend zu steigern.

Ohne ein Tool allerdings, das die Verwendung solcher Muster aktiv unterstützt, ist nicht viel gewonnen. Zu vieles beschränkt sich dann auf das fehleranfällige, manuelle Abschreiben und Kopieren von Vorlagen. Ich komme zur abschliesseden Tool-Übersicht.

## 1.5 Glossar

Ein UML-Modell wird mithilfe von spezifischen Erweiterungen (Stereotypen, Tagged Values, OCL) innerhalb einer formalen Designsprache (UML-Modellierungssprache) genauer beschrieben.

Stereotyp

Stereotypen geben kommentierend die möglichen Verwendungszusammenhänge einer Klasse, einer Assoziation oder eines Paketes an. Stereotypen klassifizieren die möglichen Verwendungen eines Modellelementes. Ein Element, beispielsweise eine Klasse, kann beliebig viele Stereotypen aufweisen.
Bsp.: «self», «import», «global»,

public: ein « + », protected: ein « # », private: ein « - » vorangestellt.

Tagged Values

Werte die in einem Modell konkret als Implementierungsanweisung zugewisen werden, wie:
PersistenceType = Transient, Derived = True, InitialValue = Ord(`A`)

OCL

Die Object Constraint Language ist eine semiformulare Sprache zur Definition von Bedingungen/Einschränkungen. Sie definiert eine standardisierte Sprache zur Beschreibung von Zusicherungen innerhalb von Modellen.

Bsp.: {FIndex < FList.Count}, {salesman.knowledgeLevel >= 5}

Plattform

Dieser Begriff ist innerhalb der MDA krititsch zu betrachten, es ist das Laufzeitsystem gemeint, dazu zählen Frameworks wie auch Bibliotheken, die dann in der spezifischen Ausprägung auf der jeweiligen Hardware-Plattform laufen. Der Generator benutzt die Plattform.

MDA Generator

Der Parser des Generator Backend interpretiert ein UML-Profil oder ein Template, erzeugt den plattformspezifischen Code via FileStreaming, scannt und identifiziert die geschützen Bereiche bevor die schon vorhandene Fachlogik in den neuen oder modifizierten Implementationsrahmen geschrieben wird.

Max Kleiner, Oktober 2003

[i] Kleiner: „Patterns konkret“, Software & Support Verlag, 2003

[ii] MDA Specifications, www.omg.org/mda/

[iii] b+m Generator Framework www.architectureware.de, b+m Informatik AG

[iv] Statemate, www.ilogix.com/products/magnum/index.cfm

[v] Kleiner: Designermodell in: Der Entwickler 1.2001

[vi] DelphiWebStart: http://sourceforge.net/projects/delphiwebstart

[vii] www.csharp-info.de