Model Driven Atchitecture in to Support of Software Product Line

Ahmed Mohammed Elsawi[1*], Shamsul Sahibuddin[2]

[1] Faculty of Computing, University Teknologi Malaysia, Johor Bahru, Malaysia.
[2] Dean of Advance Informatics School, University Teknologi Malaysia, Kuala Lumpur, Malaysia.

* Corresponding author. Tel.: +249926100000; email: elsawi@gmail.com
Manuscript submitted October 20, 2014; accepted December 16, 2014.
doi: 10.17706/ijcee.2015.v7.871

Abstract: Product management is focusing on variation description; as current and future possible market positioning, acceptable limitations, business opportunity, etc. On the other hand, the Architectural Modeling is concentrating on creation and description of software platform and how it extends and evolves to deliver and support all or any of these product management and product line scenarios. In product line engineering software features are always defined by the variability model. While the requirements specifications, architectural, design, implementation and testing are normally captured by the solution space. In this work we are utilizing the architectural modeling concept of the Model Driven Architecture (MDA) as a complementary of the software product line concerning the solution space part. Aiming to improve software quality and productivity the MDA principles has been adopt. We explicitly employed the Platform Model (PM) designed by the Entity-Attribute-Value (EAV) concept, to provision the transformation between Platform Specific Models (PSM) to the targeted platforms from side, and to manage the platforms diversity and volatility from other side. A case study presented to demonstrate the software productivity under the orthogonal relationship between the MDA and the SPL at the later solution space side.

Key words: Model driven architecture, software product line, platform model, EAV, MDA.

1. Introduction

The dynamic change in technologies drags the enterprises to a frantic race to keep up with this technology evolution. Consequently, modern software applications are regularly required to come in multiple platform support feature. But the software size and their development process are growing too complex. The thing that reflected on the quality and software frames profitability and productivity. Different software methodologies adopted software reusability to reduce the cost and time to market along with the focus on platforms diversity. The Software Product Line (SPL) and Model Driven Architecture (MDA) are examples of these methodologies, that handling the above targets each of its way. While the SPL focusing on not reinventing software applications by seeking commonalities in applications to be developed once and reused in upcoming releases in a specific domain [1], The MDA adopted the separation of concern and employed models as a keystone in software development productivity process [2].

The development process of the SPL focusing on reusability of the software specifications, architectures and components [3]. Fig. 1, represent the Domain Engineering and Application Engineering as the two main phases of the SPL development process [4]. The Domain Engineering is concerned with the domain analysis, design and implementation as the core asset developed for reuse. While the Application Engineering

concerned with the application requirements, design and implementation based on the specific customer requirements utilizing the core assets in a specific domain.

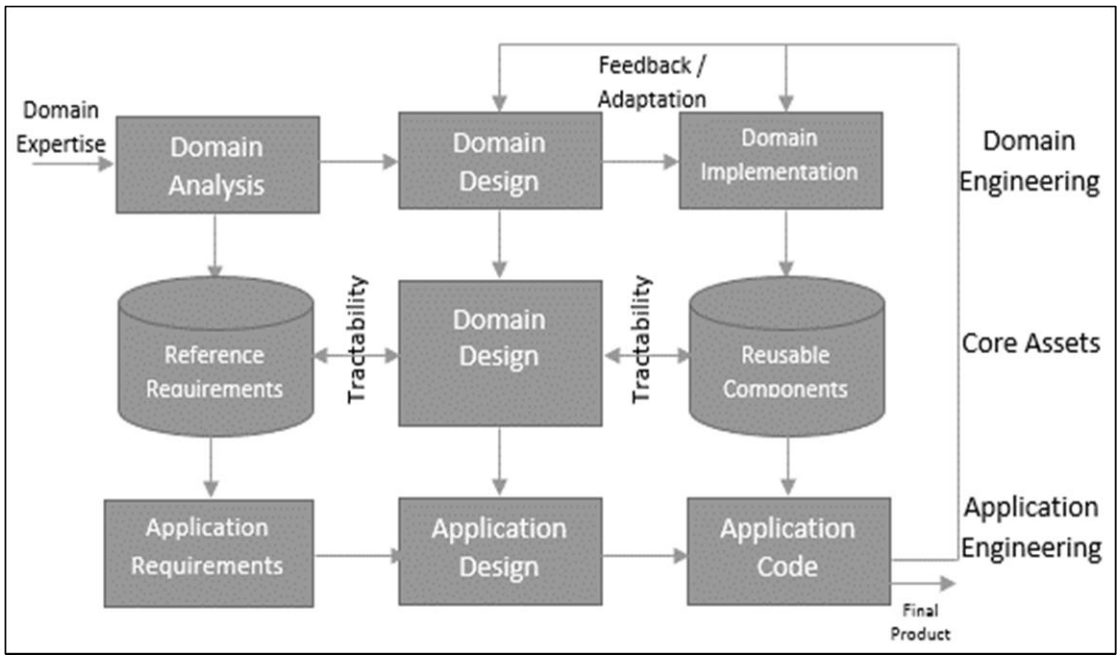

Fig. 1. Software product line process [4].

The application requirements captured by the Reference Requirements defined by the Domain Analysis. While the Application structure captured by the Reference Architecture that defined by the Domain Design. The reusable components are reused to code application. Future change and updates are managed by the bidirectional reusability [5].

From the above we can say that the Domain Engineering provide the product line scope, core asset for reuse, and a plan for software productivity. These component based assets of the SPL considered as an advantage as well as a main drawback approach as its software in a specific domain, produce applications for a specific platform. In other word, to reproduce a product to work in a different environment need a huge redevelopment and coding, since the asset components designed in the first place to address specific platforms.

The model based development lifecycle in MDA is somehow overcome this limitation of SPL, since the focus here in Architectural Modeling the concern of the business requirement is separated from the technical implementation and platform requirements. The model driven development approach divided to platform independent model (PIM) and platform specific models (PSM). Both models are working in different level of abstractions [2]. UML/MOF are a common OMG standard tools that normally used in model driven development to design models and metamodels [6], [7]. Model transformation is one of the main activity in model driven software that serve in transform high level models to low level models using model transformation tools such as Query-View-Transformation (QVT), Atlas Transformation Language (ATL) and Eclipse Model Framework (EMF).

The adoption of separation of concern and models along with model transformation, gives the MDA a great advantage on the automation of software development. Therefore, this MDA capability is enhancing the software quality and productivity in a very cost effective way comparing to the classical software development. Fig. 2, show the effectiveness of MDA lifecycle in software productivity and maintainability comparing with classical software development lifecycle.

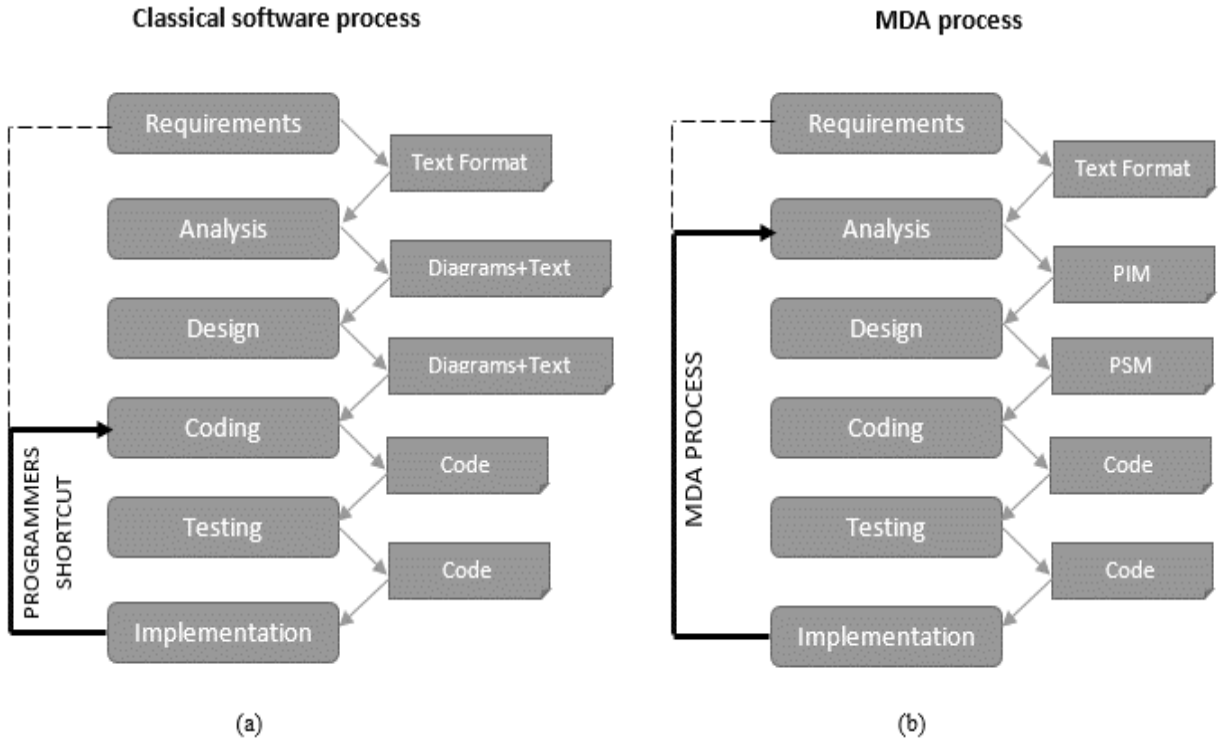

Fig. 2. Classical software development lifecycle vs. MDA lifecycle.

Fig. 2 (a) represent the classical software development process. Practically, any possible automation is taking place from the coding level downward. Consequently, changes in coding doesn't reflected to the top levels, and updates on the top levels above the coding on the other hand means a considerable effort of recoding. This is due to the fact the text and diagrams in the above layers are normally used for documentation and communication purposes [8]. The MDA process shown in Fig. 2 (b), present an automated closed loop of between levels. Where, changes in the abstract top levels can automatically be propagated to the lower ones, with minimal effort, time and cost. On the other hand, changes on the lower abstract levels can be reflected to the upper ones, by reversing the transformation from lower level of abstractions code/artifacts to the upper layers with higher level of abstraction models[9].

Similar to classical software development process, the SPL process in Fig. 1, is lacking the automation flexibility between the different levels of the development process. Where the bottom-up and top-down changes are limited cannot be propagated upward or downward respectively. For example any change in the domain engineering area require a manual update on the core asset and consequence the application engineering. This is due to the fact that the diagrams and charts adopted to descript the domain engineering doesn't exceed the documentation and communication purpose.

In this work we motivate for the MDA capabilities in to support of the software product line concept. We also illustrated how to address the platform diversity and volatility by an explicit employment of platform models. This explicit platform model inspired by the Entity-Attribute-Value concept. It was designed to support the transformation from platform specific model to targeted platform code. Next, is Section 2 we introducing the related works, and initiatives that focused on the collaborations between software product line and mode driven architecture. This is beside the work that address the issues of platform diversity in the MDA context. Section 3 show the capability of EAV in modelling a computer platform and how we utilize it to design an explicit platform model to support the model transformation from PSM models to targeted computer platform. A case study presented in Section 4. The discussion and results shown in Section 5. While the conclusion and future work provided in Section 6.

2. Related Work

The work in [10] proposed an End to End Development Engineering (E2EDE) to bridge the transformation gap between PIM and PSM. The author relied on the notion of variability in SPL to support model design decisions in PSM. In reverse, the collaboration between the two methodologies presented in [11], as they introduced an automation configuration process created on SPL, and proposed an integration of MDA platform model in a software development process. This work is closely related to ours. Where the authors suggest an explicit use of an ontology based platform model to address the platform dependencies of the model transformations and their validity to specific platforms. Using the ontology to represent the platform model comes with the limitations. In general, the automation of handling big ontology is impossible due to the number of classes an instances. This is beside the fact that the time consumed in a manual construction of ontologies is growing more complex upon the diversity and platform's data volume rapid increase. Consequently, this is reflected in the number of ontologies required to cope with this technology volatility. Although some ambitious ontology automation[12], [13] works on reducing the time and complexity, however, it is not yet mature enough to be adopted. Putting in to account that the current ontology automation methods are failing to merge different ontologies to produce new mature one [14], [15]. Another ontology platform representation limitation, is it lack flexible validators that capable to validate all different type of platform ontologies. Especially when it comes to complex inheritance relationship. On the other hand the OWL language that adopted to create the platform ontologies is also draw some limitation on ontology creation. Since, the OWL doesn't stand on a backend database during the ontology creation.

Nevertheless, is following some of their steps as we also explicitly employed the platform model. However, we used the EAV model to address the platform representation issues and to equip the model transformation with the necessary technical details required by the targeted platforms that we designed and store it in EAV designed platform model.

In SPL terminology the platform model here can be described as a core assets that can be utilized by model transformations to address different platforms.

In the next section we highlight EAV capabilities in knowledge representation and specifically in representing computer platforms models and metamodels along with the design of the explicit Platform Model.

3. Modelling of Computer Platform Using Entity-Attribute-Value

Aiming to describe platform and platform's dependencies and constrains we employed the Entity-Attribute-Value (EAV). EAV is widely used in the medical and clinical information system as a general purpose means of knowledge representation. The Attribute-value pairs concept are an esteemed way of representing information on an object, originated on 1950s on the LISP association lists [16]. An example of attribute-value pairs showing a particular student information would be:

((IndexNo A3) (Program CS101) (GPA 3.1) (Year 2012) (Status Active))

Relational databases are traditionally designed using at least first normal form (all values are atomic, with no repeating groups), and so attribute-value pairs become triples with the entity (the thing being described, identified with a unique identifier of some sort) repeating in each row of a table.

Extensible Markup Language (XML) [17] syntax is related to attribute-value pairs. XML elements, delimited within open- and close-tags for ease and accuracy of parsing, can represent either entities or attributes. They can contain sub-elements nested to arbitrary levels; sub-elements may be regarded as

attributes with complex structure. For convenience, atomic data describing an entity may also be represented within an element's open-tag as attribute-value pairs, each component of a pair being separated by an equal sign.

3.1. EAV Representation for Model and Metamodels

In this part we are presenting how we represents models and metamodels in EAV format. Fig. 3 shows our instance model that we designed by a simple State Machine design language for an application in which Passengers buy tickets at the time they obtain reservations. At check-in time they obtain boarding cards if there are still seats available. Due to over booking of flights they may be rescheduled on later flights.

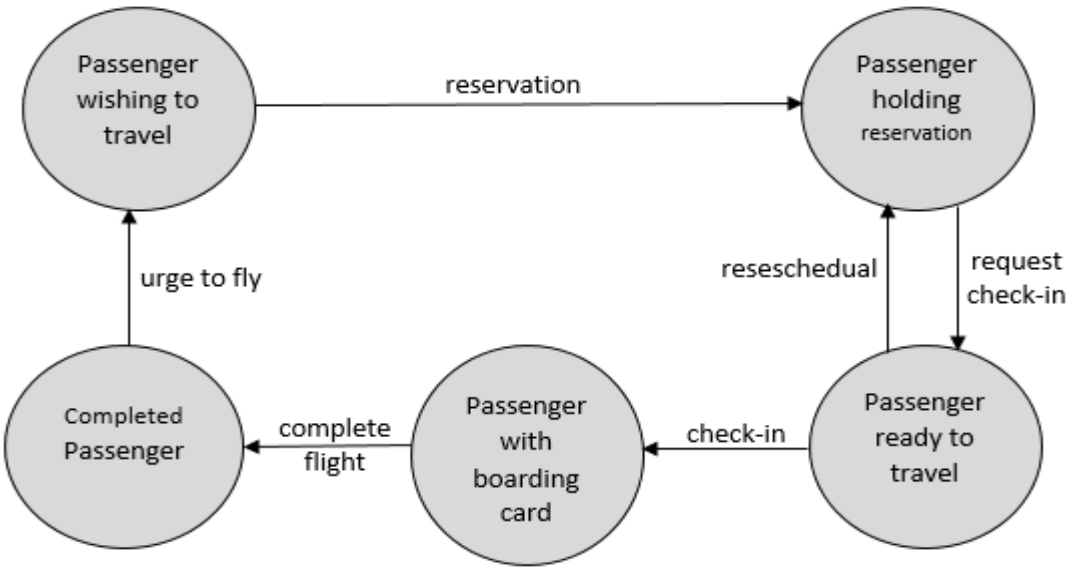

Fig. 3. A state machine model for airline passenger.

Some of the information in the Airline Passenger model is implicit. In this situation, we need to interpret the graphical objects in the diagram, which we do by consulting the documentation of the State Machine modeling language and its particular representation in this case.

Here, there are three types of object:

- States, represented by ovals, each of which has a name, represented by the text contained in the oval.
- Transitions, represented by arrows. A transition is from a source state (represented by the plain end of the arrow) to a target state (represented by the end of the arrow with an arrowhead).

Events, each of which is associated with a transition. An event is represented by a name near the arrow representing the associated transition. The diagram contains five instances of State: Passenger

A metamodel representing the concept in Fig. 3 is shown in Fig. 4, represented as a UML Classes diagram. Note that the instances in the diagram of Fig. 3 do not appear in the metamodel of Fig. 4. Note also the metaclass NamedElement, which is a superclass of the metaclasses State and Event. The states and events of Fig. 3 are all named. The metaclass NamedElement supplies an attribute name to its subclasses.

Metamodels are closely related to database schemas. Instances of the concepts specified in the model are stored in a database specified by the schemas developed from the metamodel. Fig. 5 shows the instances in the class list of Fig. 3 represented in a database whose schema is developed from the metamodel of Fig. 4.

Notice that the population of the database in Fig. 5 consists entirely of tuples of literals. Each column of each table is relational attribute of a literal type. A column in a table is ultimately derived from a literal-valued attribute in the UML Classes model of Fig. 4. We can think of the population of the database as a collection of literals organized according to the classes, associations and attributes in the Classes model.

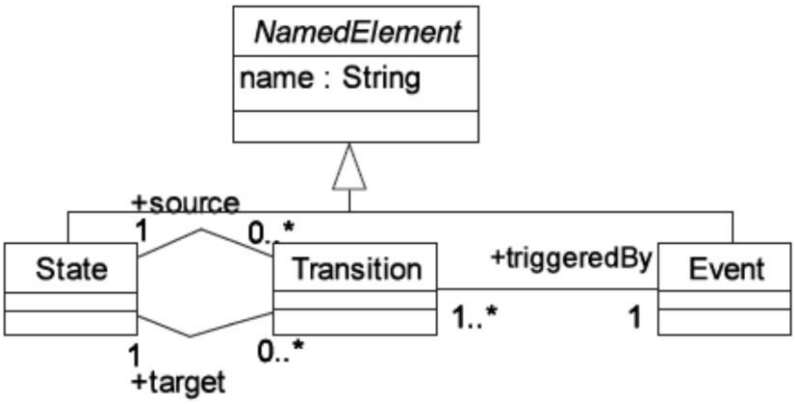

Fig. 4. UML class diagram for state machine.

State
Name
WishTravel
Completed
HoldRes
ReadyTravel
WBoardCard

Event
Name
Reservation
Reschedual
reqCheckIn
checkIn
Complete
urgeFly

Transition		
Source	Target	triggeredby
WishTravel	HoldRes	reservation
HoldRes	ReadyTravel	reqCheckIn
ReadyTravel	HoldRes	reschedual
ReadyTravel	WBoardCard	checkIn
WBoardCard	Completed	complete
Completed	WishTravel	urgeFly

Fig. 5. Airline passenger state model of Fig. 3 represented as a conventional database population.

In the same way, the instances in the Airline Passenger model of Fig. 3 can be represented as a population of a database whose schema is developed from the metamodel of Fig. 4, as shown in Fig. 6. (Abbreviation used for more space). This database is the repository of a modelling tool supporting the simple State Machine design language. Here the columns are all derived from the name attribute of the class NamedElement in Fig. 4. This conventional representation for the database tables State and Event, where they have only one column, name. The table Transition has three columns, all foreign keys. Two are derived from the name attribute of the class State and one from the name attribute of the class Event. Without the attribute name in NamedElement, it would be impossible to create a repository schema that would record the Airline Passenger model of Fig. 3.

A further issue is that a relational schema requires that for each table certain attributes be declared to be the key for the table. That is, a row in the table can be identified by looking at the values of the key attributes. Knowing the values of the key attributes, we can look in the table to find the values of the other attributes in the row. Some metamodeling languages allow the specification of identifiers. Entity-Relationship Modeling and Object-Role Modeling both support identifiers. UML, however, does not [18]. If UML is used as the metamodeling language, then additional information must be supplied to designate some attributes in the repository schema to be keys.

In the STM repository of Fig. 3, the tables State and Event both have the attribute name as key, while the Transition table has a key composed of the three attributes source, target and triggeredBy (see Fig. 5).

Once we have a schema and a population for an application, we can use the query language associated with the database system to make queries about the population. Queries are typically about the semantics of the application. Nevertheless, any change on the metamodel in Fig. 4 should by reflected on its instance model in Fig. 3 and consequently in the database in Fig. 5. However, because of the conventional database structure a Data Definition Language (DDL) statements should be used. For example to add new attribute to the table Event or State an Alter table statement should be employed. Which normally done by the model designer who's not necessary the one who is doing the development. On the other hand, most of the modelling tools does not allowing any changes on their main metamodel on which they developed based on it. To overcome this limitation a dynamic structure employed to replace the conventional schema in Fig. 5 by EAV structure in Fig. 6. The open structure of EAV treat all the tables in the conventional schema as a tuple entry in a single EAV table. The thing that gives more control in managing models dynamicity, upgrade and maintenance.

ENTITY	ATTRIBUTE	VALUE_
EVENT	NAME	checkIn
EVENT	NAME	Complete
EVENT	NAME	reqCheckIn
EVENT	NAME	Reschedule
EVENT	NAME	Reservation
EVENT	NAME	urgeFly
STATE	NAME	Completed
STATE	NAME	HoldRes
STATE	NAME	ReadyTravel
STATE	NAME	WBoardCard
STATE	NAME	WishTravel
TRANSITION	SOURCE	Completed
TRANSITION	SOURCE	HoldRes
TRANSITION	SOURCE	ReadyTravel
TRANSITION	SOURCE	ReadyTravel
TRANSITION	SOURCE	WBoardCard
TRANSITION	SOURCE	WishTravel
TRANSITION	TARGET	Completed
TRANSITION	TARGET	HoldRes
TRANSITION	TARGET	HoldRes
TRANSITION	TARGET	ReadyTravel
TRANSITION	TARGET	WBoardCard
TRANSITION	TARGET	WishTravel
TRANSITION	TRIGGEREDBY	checkIn
TRANSITION	TRIGGEREDBY	complete
TRANSITION	TRIGGEREDBY	reqCheckIn
TRANSITION	TRIGGEREDBY	reschedule
TRANSITION	TRIGGEREDBY	reservation
TRANSITION	TRIGGEREDBY	urgeFly

Fig. 6. Airline passenger state model of Fig. 3 represented in EAV database population.

Structure-oriented queries are important in Modelling tool applications. For example, a state machine can have an initial state (a state with no transitions in) or a final state (a state with no transitions out). These states can be identified respectively by the following two views:

```
CREATE VIEW InitialState(StateName) AS(
        SELECT A.Value_ FROM EAV A
                Where A.ENTITY = 'STATE' AND A.ATTRIBUTE = 'NAME'
                AND
                A.Value_ NOT IN (
                SELECT B.Value_ FROM EAV B
                        WHERE
                                B.ENTITY = 'TRANSITION'
                                AND
                                B.ATTRIBUTE = 'TARGET'))

CREATE VIEW FinalState(StateName) AS(
          SELECT A.Value_ FROM EAV A
                  Where A.ENTITY = 'STATE' AND A.ATTRIBUTE = 'NAME'
                  AND
                  A.Value_ NOT IN (
                  SELECT B.Value_ FROM EAV B
                          WHERE
                                  B.ENTITY = 'TRANSITION'
                                  AND
                                B.ATTRIBUTE = SOURCE))
```

The above two views return no data because the state machine in Fig. 3 is cyclic. Hence, we interested to validate whether our state machine design is entirely cyclic, with neither initial nor final states.

```
CREATE VIEW CyclicModel(Cyclic) AS
        SELECT "Cyclic" FROM State WHERE
        NOT EXISTS SELECT * FROM InitialState
        AND
        NOT EXISTS SELECT * FROM FinalState
```

In particular, Modelling Tool repositories are intended to store designs, which are often expressed in graphical languages (like UML). The two-dimensional nature of graphical languages makes it relatively easy to have a design language where the design concepts are expressed as a complex structures. These complex structures generally have formation rules (Constrains), which can be checked by structural queries. Structural queries therefore are more important for modelling tools than for general database applications.

An example of a design language (metamodel) with complex structures having constrains is our simple State Machine language of Fig. 4. An instance of Transition is necessarily linked to two instances of State and one instance of Event. A structural query whose result is violations of this constrain is as below:

```
SELECT * FROM EAV A WHERE
A.ENTITY = 'TRANSITION' AND
   NOT EXIST(
       SELECT * FROM EAV B WHERE B.ENTITY = 'STATE'
           AND B.ATTRIBUTE = 'NAME'
           AND B.VALUE_ IN
          (SELECT B1.VALUE_ FROM EAV B1 WHERE
```

```
            B1.ENTITY = 'TRANSITION'AND
            B1.ATTRIBUTE ='SOURCE')
    AND
    SELECT * FROM EAV C WHERE C.ENTITY = 'STATE'
        AND C.ATTRIBUTE = 'NAME'
        AND C.VALUE_ IN
          (SELECT C1.VALUE_ FROM EAV C1 WHERE
              C1.ENTITY = 'TRANSITION'AND
              C1.ATTRIBUTE ='TARGET')
    AND
    SELECT * FROM EAV D WHERE D.ENTITY = 'EVENT'
        AND D.ATTRIBUTE = 'NAME'
        AND D.VALUE_ IN
          (SELECT D1.VALUE_ FROM EAV
              D1 WHERE D1.ENTITY = 'TRANSITION'AND
              D1.ATTRIBUTE ='TRIGGEREDBY')
)
```

Additional constrains can be added in to a given design, for example that there be exactly one initial state and exactly one final state, or that there be no isolated states.

Some modeling languages allow constraints to be represented by annotations on the model, but it may not tell a designer how to concretely represent a design. For example, the UML model of Fig. 4 does not tell the designer enough to be able to represent the design of the Airline Passenger state model so that it looks like Fig. 3. To do this, the conceptual model must be augmented by some rendering conventions. However, we are implementing this by joining both, model and metamodel presented in Fig. 3 and Fig. 4 respectively in a single EAV structure, that we introduced in [19] as a novel open source metamodel concept.

In the next part we presented an EAV representation for a computer platform and a platform model designed by EAV.

3.2. EAV Representation for Computer Platform

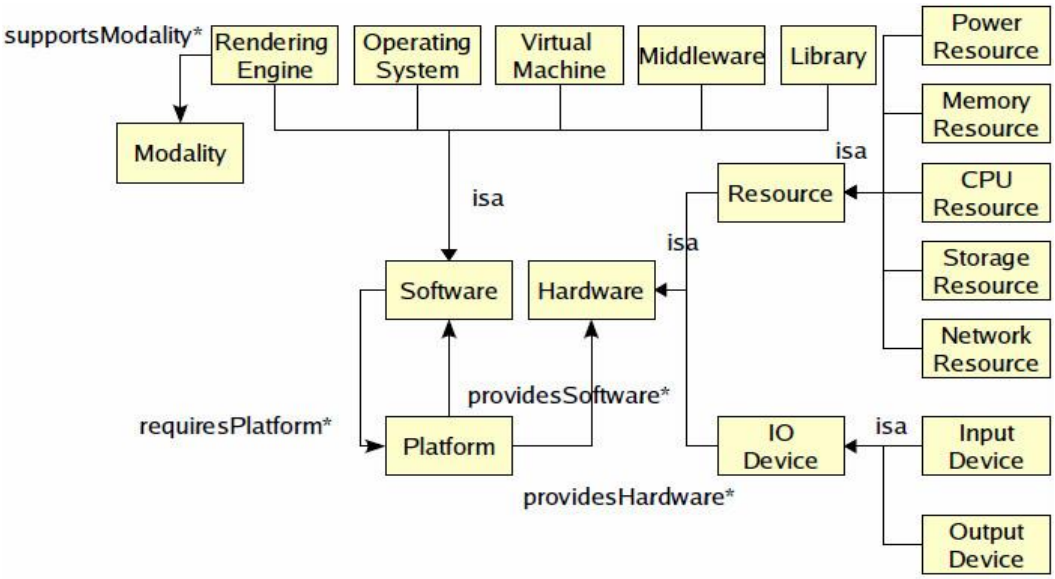

Fig. 7. A partial view of a computer platform description presented by [11].

In Section 3.1 we show EAV capabilities in representation of models and metamodels. Its knowledge representation capability and open structure make it suitable for modeling and representing computer platform. The open structure gives EAV a flexibility to cope with the platform diversity and volatility. Our work in [19] demonstrated the possibility of merging models with different level of abstraction in a single EAV repository. This capability open the door for more technology merger and integration flexibility. On the other hand it gives valuable support to both SPL and MDA approaches. Since both are focusing on speed up and cost cut the software development process. With consideration to satisfy the technology diversity factor. In this section we adopted the computer platform description provided by [11]. They provide an ontology based platform vocabulary that we utilized and presented in EAV model. Fig. 7 show a partial view of their platform description.

Each platform component in the above figure, can be represented as an Entity with its associated attributes and values. For example the virtual machine component presented as Platform.software.virtualmachine. The "isa" is a subsumption relationship that indicate that the group of Virtual Machines subsumes the group of software. The Entity associated attributes and values, list out the information and futures provided by this Entity. This is along with its correspondent dependencies and constrains that govern its interactivities with other platform components. Table 1 illustrate a partial EAV representation for Fig. 7. For sake of space saving and simplicity we narrow down the amount of attributes, so we can present the whole concept above.

Table 1. A Partial of a High Level EAV Representation for A Platform Presented in Fig. 7

ENTITY	ATTRIBUTE	VALUE
Platform	Relationship.type	Multiplicity.provideHardware.Platform.hardware
Platform	Relationship.type	Multiplicity.provideSoftware.Platform.software
Platform.software	Relationship.type	Multiplicity.requirePlatform.Platform
Platform.software.library	Relationship.type	Subsumption.isa.Platform.software
Platform.software.middleware	Relationship.type	Subsumption.isa.Platform.software
Platform.software.virtualmachine	Relationship.type	Subsumption.isa.Platform.software
Platform.software.operatingsystem	Relationship.type	Subsumption.isa.Platform.software
Platform.software.renderingengine	Relationship.type	Subsumption.isa.Platform.software
Platform.software.renderingengine	Relationship.type	Multiplicity.supportsModality.Platform.software.renderingengine
Platform.hardware.resource	Relationship.type	Subsumption.isa.Platform.hardware
Platform.hardware.IODevice	Relationship.type	Subsumption.isa.Platform.hardware
Platform.hardware.resource.powerresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.resource.memoryresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.resource.CPUresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.resource.storageresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.resource.networkresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.resource.powerresource	Relationship.type	Subsumption.isa.Platform.hardware.resource
Platform.hardware.IOdevice.inputdevice	Relationship.type	Subsumption.isa.Platform.hardware.IOdevice
Platform.hardware.IOdevice.outputdevice	Relationship.type	Subsumption.isa.Platform.hardware.IOdevice

The Virtual Machine is prefix by Software to indicate it refers to the Software, where the '.' employed as a name space delimiter for easy navigation and information retrieval. Similar to our EAV representation in Section 3.1, the platform formation rules and Constrains, will be checked by structural queries as well. For example, to satisfy the fact that each software platform is in a cyclical relationship with software and hardware platform.

```
SELECT * FROM    EAV_PM AS B
WHERE
B.ENTITY IN (SELECT SUBSTR( A.VALUE_ , -8, 8 ) FROM    EAV_PM AS A
                        WHERE
                        B.ENTITY = SUBSTR( A.VALUE_ , -8, 8 )
                )
```

The highlighted output results in the below XML format illustrated that for each software platform there are multiple hardware and software platform involved. Having the constrains among other EAV PM results on XML, open the door for more integration possibilities with different tools in different branch streams concerned with model and software development automation and code generation.

```
<database name="test">
<!--    Table eav_pm    -->
<table name="eav_pm">
<column name="ENTITY">Platform</column>
<column name="ATTRIBUTE">Relationship.type</column>
<column name="VALUE_">Multiplicity.provideHardware.Platform.hardware</column>
</table>
<table name="eav_pm">
<column name="ENTITY">Platform</column>
<column name="ATTRIBUTE">Relationship.type</column>
<column name="VALUE_">Multiplicity.provideSoftware.Platform.software</column>
</table>
</database>
```

The high abstract level platform representation in Fig. 7, can be extended to detailed lower levels of abstraction. For example the Java Runtime Environment (JRE) in Fig. 8 come with Java Virtual Machine.

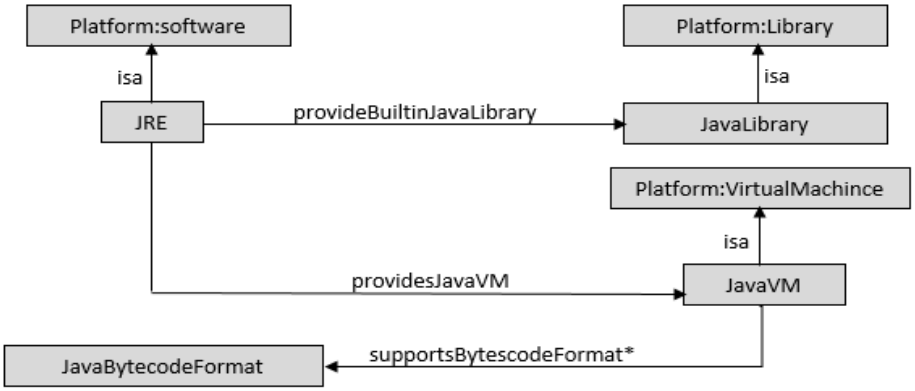

Fig. 8. An ontology fragment describe a Java Runtime [11].

Each version of Java virtual Machine has its own byte code format. In general java has different releases, each of which addressing specific technology. The major different between these releases is mainly in the build in libraries. J2me libraries addressing mobile platform technologies, while the J3EE releases

addressing different technologies through its build in APIs. In Table 2 is a part from Table 1 (for space saving). The "Platform.software.virtualmachine.JRE" is a new entity associated with its attributes values pair. For each JRE there is a dedicated virtual machine associated with build in libraries. These libraries depending on the Java specifications for the particular JRE. For example the libraries of the J2me PP 1.1 is different than the one work for J2EE. Accordingly, Table 2 show a partial description to the EAV representation of Fig. 8. It is important to mentioned that this representation is in the same EAV platform repository in Table 1 that we separated for sake of space saving.

The structural queries will also be used to set constrains and to retrieve the required API and/or technical details of the targeted platform

```
SELECT value_ FROM EAV_PM WHERE ENTITY = 'Platform.software.JRE'
```

The above code is a straight "SELECT" statement to retrieve the pair value associated to "Platform.software.JRE" Entity. By enquiring each of the highlighted entities we will get all the APIs beside the constrain we presented in the above EAV_PM example. Where the "Multiplicity.provideSoftware.Platform.software" and "Multiplicity.provideSoftware.Platform.hardware" entities are retrieved as a dependency constrain required by "Platform.software.JRE". Similarly, the "Platform.software.Library.JavaLibrary" and "Platform.software.VirtualMachine.JavaVM" can be enquired to drill down more information and/API associated to them.

1) "Subsumption.isa.Platform.software"
2) "Multiplicity.provideBuildinJavaLibrary.Platform.software.Library.JavaLibrary"
3) "provideJavaVirtualMachine.Platform.software.VirtualMachine.JavaVM".

Table 2. Partial EAV Representation of the Ontology in Fig. 8

ENTITY	ATTRIBUTE	VALUE
Platform.software.JRE	Relationship.type	Subsumption.isa.Platform.software
Platform.software.JRE	Relationship.type	Multiplicity.provideBuildinJavaLibrary.Platform.software.Library.JavaLibrary
Platform.software.JRE	Relationship.type	provideJavaVirtualMachine.Platform.software.VirtualMachine.JavaVM
Platform.software.VirtualMachine.JavaVM	Relationship.type	Subsumption.isa.Platform.software.virtualmachine
Platform.software.VirtualMachine.JavaVM	Relationship.type	Multiplicity.supportBytecodeFormat.JavaByteCodeFormat
Platform.software.Library.JavaLibrary	Relationship.type	Subsumption.isa.Platform.software.Library

In the next part we provide a briefing about an Eclipse Modeling Framework.

4. Eclipse Modeling Framework (EMF)

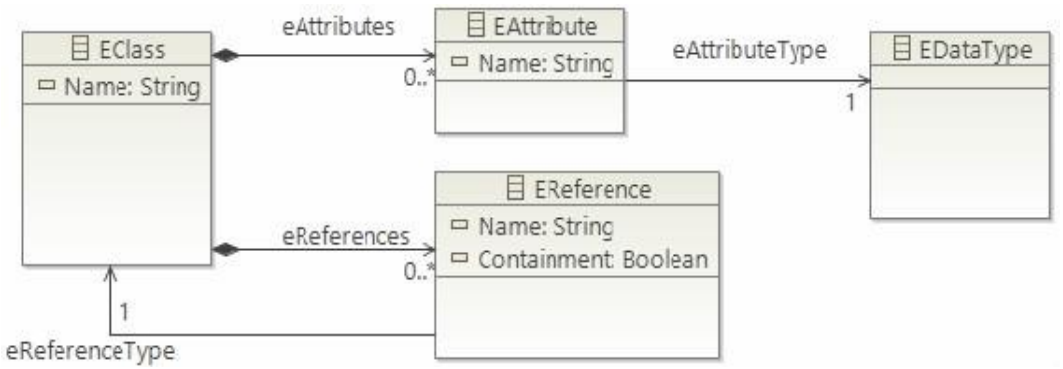

Fig. 9. A fragment from Ecore metamodel.

The Eclipse Modeling Framework is conforming with the OMG Meta Object Facility (MOF) standard for metamodel definition. The EMF metamodelling language called Ecore. Fig. 9 presenting a fragment of the Ecore Metamodel.

The capital E in the Ecore used to mark the Ecore classes from UML classes. Similarly, the Ereference and Eattribute . While the Eclass represent the classes the Ereference represent the association between classes. The Eattribute name the properties of each class and the attribute type address by the Etype.

Beside the Ecore, the EMF has another metamodel called Genmodel. While the Ecore handle the information about the defined classes. The Genmodel, provide the necessary information required from code generation.

Among other transformation tools we adopted the RMF for its modelling capability where a domain model is visibly established. This is beside the notification feature the EMF provided upon model modification. Also, the EMF managing changes in models upon object creation through its notification feature. Consequently, applications will be immune from discrete classes' employment. On the other hand, the targeted code (Java code) can be generated when desired from the source model.

In the next part a case study introduced to demonstrate the MDA capabilities in software productivity and to show the transformation from PSM to Java code utilizing the explicit EAV platform model. The EMF version adopted to implement this case study is Eclipse Juno Service Release 2[1].

5. Website Factory Case Study

The MDA principles has been adopted in this case study to build a website factory. In SPL terminology we are building a product line to produce websites. The eclipse JUNO is equipped with a visual editor that allow us to create Ecore diagrams. Our Ecore model in this case study called "webpage.ecore". The UML class diagram shown in Fig. 10, define 4 classes (Web, Webpage, Category and Article).

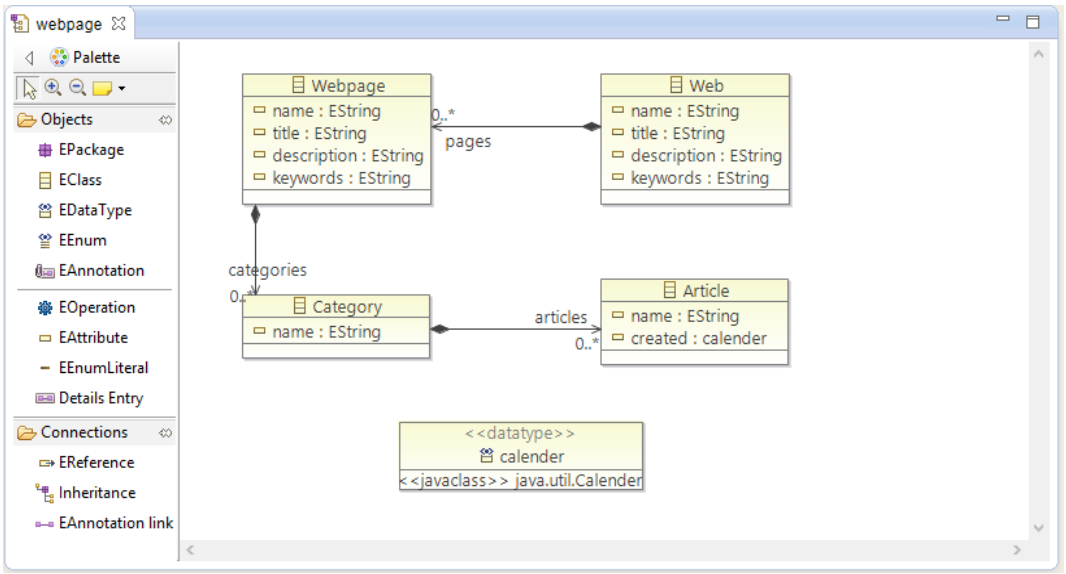

Fig. 10. A fragment from a website class diagram.

Unlike the other attributes, we assigned the attribute "created" in the "Article" class to a user defined "calender" type defined in an Edatatype named "calendar" with type "java.util.Calendar". The saved diagram stored in "webpage.ecore" in the format shown in Fig. 11. This feature open the door for including a precompiled core assets as user defined data type.

[1] http://www.eclipse.org/downloads/packages/release/juno/sr2

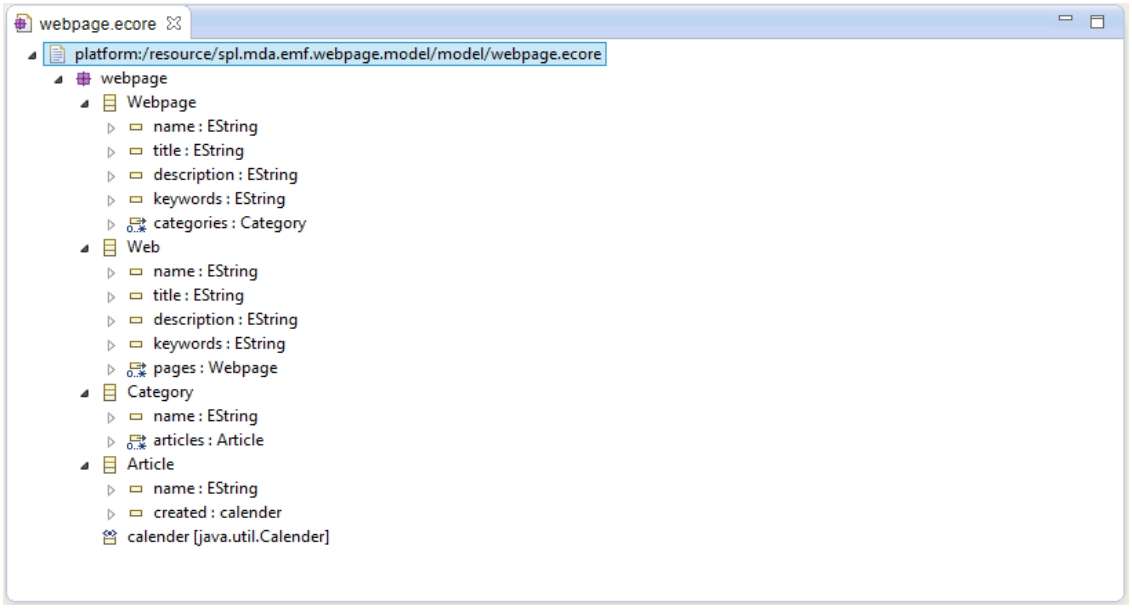

Fig. 11. Part of the "webpage.ecore" model's file format.

As we mentioned in the previous section the EMF supported beside the Ecore metamodel, the "genemodel" metamodel. The genmodel should contain all the information concerned with the code generation. Through the EMF Generator Model, we generate a new version based on the "webpage.ecore" model called "webpage.genmodel". This step done by the EMF generator Model. Fig. 12 show the "webage.genmodel" file format.

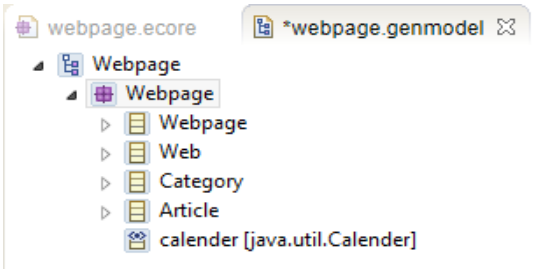

Fig. 12. Sample from the "webpage.genmodel" file format.

The both model files, the ".ecore" and ".genmodel" are required to generate the java code. By write click on the webpage node in the webpage.genmodel file (see Fig. 13) we get the following three files: The "spl.mda.emf.webpage.model.webpage" which representing the interfaced and the factory to great the Java classes. While the second is the "spl.mda.emf.webpage.model.webpage.impl" which holding the concrete code of the webpage model. The last is file is the "spl.mda.emf.webpage.model.webpage.util" which act as the adaptor factory.

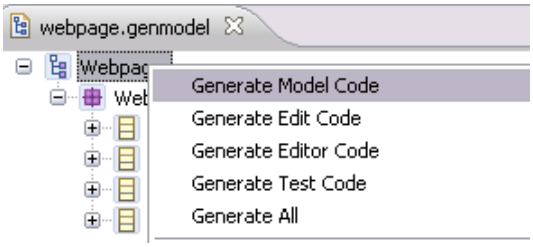

Fig. 13. Show how to generate the website concrete Java code.

In the next step we creating an editor to our generated website model. Similar to the step shown in Fig. 13, but this time will choose "Generate edit code" and then "Generate editor code" respectively. This is resulting to a ".editor" plug-in that can be run in a new eclipse instance as eclipse application. In the new runtime eclipse instance we have the leverage of using the "Example EMF Model Creation Wizards" where we choose our "Webpage Model". Then we can create an object for each of our model classes as shown in Fig. 14.

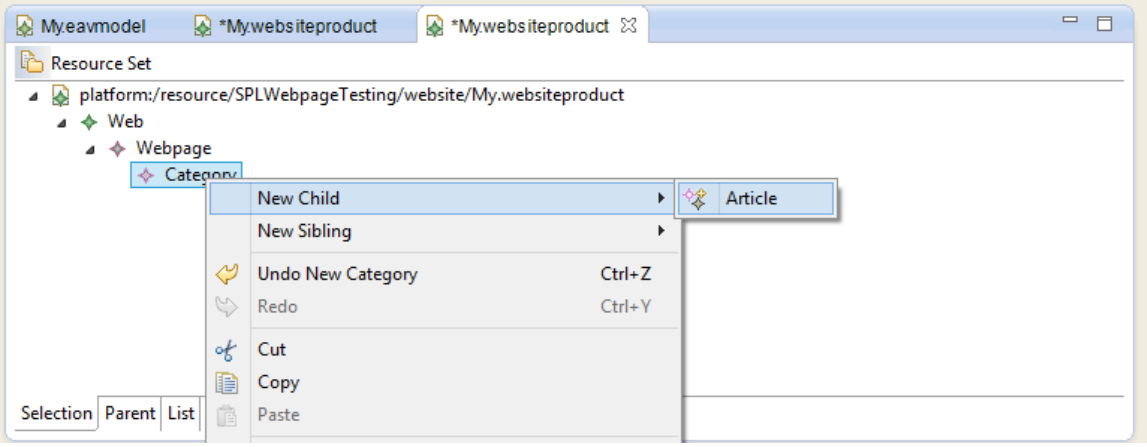

Fig. 14. Example of creating an object from webpage and category classes.

In the above example we straight forward generated an executable website java code form a website UML class diagram model. There is no employment the platform model here since our targeted platform in a normal personal computer. But if we want to extend this website to be browsed in a mobile platform. The code need to be adjusted to suite for mobile browsing.

To handle this issue we utilize our EAV platform model to update the model transformation with the necessary information to adjust our website to be brows in a mobile environment. Since the EAV platform is already updated with different platform's information up to the concrete liberary code and instead of start the transformation from scratch, we retrieve the required information from the EAV platform model as described in Section 3.2. The required code lies under the library part. So we need to retrieve the mobile browsing requirement as per below code, by retrieving the content of the mobile library API from EAV PM.

```
SELECT VALUE_ FROM (
SELECT * FROM EAV_PM A
        WHERE A.Entity = 'Platform.software.Library.JavaLib.Mobile.MobileBrowes'
            AND
        A.ATTRIBUTE = 'LIB.Type.Smartphone.General'
            AND
        A.VALUE_ = 'API.Platform.software.Library.JavaLib.Mobile.MobileBrowes') AS B
WHERE
  B.ATTRIBUTE = 'LIB.API.Content'
```

The below code is a partial API content for mobile browsing, resulting of the above query.

```
public class SmartPhoneBrowsingInfo
{
    //Stores some info about the browser and device.
    private String BrowerAgentInfo;

    //Stores info about what content formats the browser can display.
```

```
    private String httpAccept;

    //       strings that list out smart phone device's capabilities.

  public static final String deviceType = "SmartphoneBrand";

    //The constructor. Initializes several default variables.
    public SmartPhoneBrowsingInfo(String BrowerAgentInfo, String httpAccept) {
            if (BrowerAgentInfo != null) {
                this.BrowerAgentInfo = BrowerAgentInfo.toLowerCase();
            }
            if (httpAccept != null) {
                this.httpAccept = httpAccept.toLowerCase();
            }
     }

    //**************************
    //Returns the contents of the broweser Agent value, in lower case.
    public String getBrowerAgentInfo()
    {
          return BrowerAgentInfo;
    }

    //**************************
    // Detects if the current device is a SmartphoneBrand.
    public boolean detectSmartphoneBrand ()
    {
            if (BrowerAgentInfo.indexOf(device SmartphoneBrand) != -1 && !detectsmartphone()) {
                return true;
            }
            return false;
    }

}
```

The ".Impl" files need to be updated by the above code. This code will be imported as an API retrieved from our "EAV_PM" platform model. The following code illustrate how this update will take place in the ".Impl" files starting with the "spl.mda.emf.webpage.model.webpage.impl". Below is a sample code of the ".Impl" file, after updating the mobile browsing information (see the highlighted code).

```
package spl.mda.emf.webpage.model.webpage.impl;


import eav_pm.lib.javalib.mobile.mobilebrowse.SmartPhoneBrowsingInfo

import java.util.Calender;
import org.eclipse.emf.ecore.EAttribute;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.EDataType;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.EReference;

import org.eclipse.emf.ecore.impl.EPackageImpl;

import spl.mda.emf.webpage.model.webpage.Article;
import spl.mda.emf.webpage.model.webpage.Category;
import spl.mda.emf.webpage.model.webpage.Web;
```

```
import spl.mda.emf.webpage.model.webpage.Webpage;
import spl.mda.emf.webpage.model.webpage.WebpageFactory;
import spl.mda.emf.webpage.model.webpage.WebpagePackage;

/**
 * <!-- begin-user-doc -->
 * An implementation of the model <b>Package</b>.
 * <!-- end-user-doc -->
 * @generated
 */
public class WebpagePackageImpl extends EPackageImpl implements WebpagePackage {

		private EClass webpageEClass = null;

		private EClass webEClass = null;

		private EClass categoryEClass = null;

		private EClass articleEClass = null;
		/**
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @generated
		 */
		private EDataType calenderEDataType = null;

		/**
		 * Creates an instance of the model <b>Package</b>, registered with
		 * {@link org.eclipse.emf.ecore.EPackage.Registry EPackage.Registry} by the package
		 * package URI value.
		 * <p>Note: the correct way to create the package is via the static
		 * factory method {@link #init init()}
, which also performs
		 * initialization of the package, or returns the registered package,
		 * if one already exists.
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @see org.eclipse.emf.ecore.EPackage.Registry
		 * @see spl.mda.emf.webpage.model.webpage.WebpagePackage#eNS_URI
		 * @see #init()
		 * @generated
		 */
		private WebpagePackageImpl() {
			super(eNS_URI, WebpageFactory.eINSTANCE);
		}

		/**
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @generated		*/
// Initialize classes and features; add operations and parameters
			initEClass(webpageEClass, Webpage.class,
"Webpage", !IS_ABSTRACT, !IS_INTERFACE, IS_GENERATED_INSTANCE_CLASS);
			initEAttribute(getWebpage_Name(), ecorePackage.getEString(), "name", null, 0, 1,
Webpage.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
			initEAttribute(getWebpage_Title(), ecorePackage.getEString(), "title", null, 0, 1,
Webpage.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
```

```
                initEAttribute(getWebpage_Description(), ecorePackage.getEString(), "description",
null, 0, 1, Webpage.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEAttribute(getWebpage_Keywords(), ecorePackage.getEString(), "keywords",
null, 0, 1, Webpage.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEReference(getWebpage_Categories(), this.getCategory(), null, "categories", null,
0, -1, Webpage.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE,
IS_COMPOSITE, !IS_RESOLVE_PROXIES, !IS_UNSETTABLE, IS_UNIQUE, !IS_DERIVED, IS_ORDERED);

                initEClass(webEClass, Web.class, "Web", !IS_ABSTRACT, !IS_INTERFACE,
IS_GENERATED_INSTANCE_CLASS);
                initEAttribute(getWeb_Name(), ecorePackage.getEString(), "name", null, 0, 1,
Web.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEAttribute(getWeb_Title(), ecorePackage.getEString(), "title", null, 0, 1,
Web.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEAttribute(getWeb_Description(), ecorePackage.getEString(), "description", null,
0, 1, Web.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEAttribute(getWeb_Keywords(), ecorePackage.getEString(), "keywords", null, 0, 1,
Web.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEReference(getWeb_Pages(), this.getWebpage(), null, "pages", null, 0, -1,
Web.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE,
IS_COMPOSITE, !IS_RESOLVE_PROXIES, !IS_UNSETTABLE, IS_UNIQUE, !IS_DERIVED, IS_ORDERED);

                initEClass(categoryEClass, Category.class, "Category", !IS_ABSTRACT, !IS_INTERFACE,
IS_GENERATED_INSTANCE_CLASS);
                initEAttribute(getCategory_Name(), ecorePackage.getEString(), "name", null, 0, 1,
Category.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEReference(getCategory_Articles(), this.getArticle(), null, "articles", null, 0, -1,
Category.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE,
IS_COMPOSITE, !IS_RESOLVE_PROXIES, !IS_UNSETTABLE, IS_UNIQUE, !IS_DERIVED, IS_ORDERED);

                initEClass(articleEClass, Article.class, "Article", !IS_ABSTRACT, !IS_INTERFACE,
IS_GENERATED_INSTANCE_CLASS);
                initEAttribute(getArticle_Name(), ecorePackage.getEString(), "name", null, 0, 1,
Article.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);
                initEAttribute(getArticle_Created(), this.getcalender(), "created", null, 0, 1,
Article.class, !IS_TRANSIENT, !IS_VOLATILE, IS_CHANGEABLE, !IS_UNSETTABLE, !IS_ID,
IS_UNIQUE, !IS_DERIVED, IS_ORDERED);

                // Initialize data types
                initEDataType(calenderEDataType, Calender.class, "calender",
IS_SERIALIZABLE, !IS_GENERATED_INSTANCE_CLASS);

                // Create resource
                createResource(eNS_URI);
        }

} //WebpagePackageImpl
```

Now the ".genmodel" need to be reload after the above updates (see Fig. 15). After we reload the new updated model, the webpage can be regenerated. But this time the website is suite to be browse in a mobile environment.

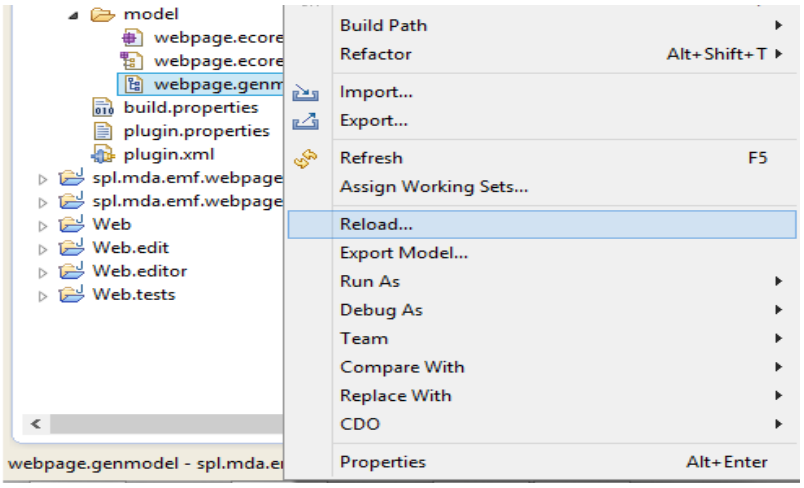

Fig. 15. Showing how to reload the ".genmodel".

It worth to mention that the changes was only limited to the ".genmodel", while the ".ecore" model remain with no changes asst. This operation can be fully automated process with minimal manual interference.

6. Discussion and Results

In this work we demonstrated the possible collaboration between the MDA and SPL. While the MDA aiming to design and build software product that can be run in multiple platforms with minor engineering, the SPL is focusing on a fast use of transformed configurations of the very same product.

However, on the highest possible level, both approaches are aiming to enhancing the software quality, productivity, interoperability, cost and time to market. But in practice, both approaches operate on different levels to achieve the same goals. The MDA adopting models in a higher abstraction levels to produce software, avoiding hard coding, but the SPL approach is dealing perfectly when applied directly on code level to build a core asset APIs. Yet, in SPL it is vital to find out the differences and commonalities between the product parts. Furthermore, it is important to identify the relationship between these parts and formalize its semantic. That's often the weakest link in SPL approaches and tools.

Despite of how does each approach work, we provided a full demonstration of how can the MDA principles applied to support the software product line approach.

EAV act as a means of knowledge representation that we utilizes to design EAV platform model. The advantage of EAV representation over the ontology platform representation is that the open structure of EAV allow for representing multiple platforms, with different level of details up to the code level. This is beside the flexibility and the dynamicity of EAV structure in defining and representing any upcoming new platform or modifying the current one.

On the other hand, the possibility of presenting EAV platform model in XML format is giving an integration room with different systems, tools and approaches. For example EAV platform can be used as a core asset repository in any software product line. The platform information and APIs can be stored and retrieved from it. Consequently, Both MDA and SPL can share this platform model as a centric area of collaboration, integration and information exchange. The SPL core assets can be stored in the EAV Platform model adopting the same semantic presented in this work. This will positively reflected on the number of platforms that the model transformation can support. This collaboration between the two approaches will complement each other and will enhance the quality and productivity of software. From other prospective the possibility of EAV platform XML format can be updated by the platform vendor's website feeds (RSS). This way the platform model will be up to day with new platform releases or versions. Consequently, the software productivity will rapidly increase supporting a great number of platform with minimal changes

and cost effective. While the coding effort can be shifted in more architectural and design work to improve the software quality.

On top of the above, the explicit employment of EAV platform model a great support to the model evolution: with proper mechanisms and tools like the EMF in place, there is a flexibility to ensure that models will work, open in editors, produce the code etc. with the newer metamodel too (e.g. updates automatically the models to the new metamodel).

There are also other advantages like faster metamodel/language development, easier management, possibility to couple various generators based on the metamodel together, etc. The think that support software product line productivity as well.

The limitation of the EAV Platform model is inherited from EAV representation drawbacks. Where a considerable up-front programming is needed to do many tasks that a conventional architecture would do automatically. Moreover, such programming needs to be done only once, and availability of generic EAV tools could remove this limitation. Also, for bulk retrieval EAV design is considered less efficient than a conventional structure. Consequently, performing complex attribute-centric queries, which are based on values of attributes, and returning a set of objects is both significantly less efficient as well as technically more difficult.

7. Conclusion and Future Work

Computer platforms are combining a set of features and component that allow to formally control different functions and contexts (Hardware/software activities) that normally limited to a particular stream of technology. Having a capability of representing multiple platforms for different technologies in different levels of abstraction, in a single EAV repository, is dramatically increasing the quality and the productivity of the software development process. In other word we show how the MDA principles can contribute positively in software quality and productivity in general, and in particular how the collaboration between the MDA and SPL can be useful for both approaches.

The EAV platform introduced in this work as a novel approach that explicitly employed to support the transformation from PSM models to java code. The EMF used for modeling, transformation and code generation. The Website factory case study presented to show the MDA capability in producing end to end software product. The website designed for desktop browsing. However, we put the EAV platform in action to update the model transformation by retrieving the mobile browsing information from EAV platform model. The model reloaded and regenerated in a mobile browse friendly version with minimal interference and modification.

In the near future the focus will be on the automatic update of the EAV platform. Where the Domain Specific Language will be explored as a solution to produce a user friendly interface that allow easy update to EAV platform model. Also, we intend to create a plug-in to automatically update EAV platform model from the vendor's website feeds.

More case studies will be implemented to target different platforms other than java for more testing and to generalization.

Acknowledgment

The authors would like to express their deepest gratitude to Universiti Teknologi Malaysia (UTM) for their financial support under Research University Grant Scheme.

References

[1] Pohl, K., Böckle, G., & Linden, F. V. D. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. Springer.

[2] Frankel, D. S. (2003). Model Driven Architecture Applying MDA: Wiley.
[3] Clements, P., & Northrop, L. (2002). Software Product Lines. Addison-Wesley Boston.
[4] Northrop, L. M. (2002). SEI's software product line tenets. *Software*, *19(4)*, 32-40.
[5] Heymans, P., Trigaux, J. C., & Objectif, F. E., (2003). Software product lines: State of the art.
[6] OMG Document. (2001). Meta Object Facility (MOF) v1.3.1.
[7] OMG Document. (2001). Unified Modeling Language v1.4.
[8] Dobing, B., & Parsons, J. (2008). Dimensions of UML diagram use: a survey of practitioners. *Journal of Database Management*, *19*, 1-18.
[9] Czarnecki, K., & Helsen, S., (2003). Classification of model transformation approaches. *Proceedings of the 2nd Workshop on Generative Techniques in the Context of the Model Driven Architecture* (pp. 1-17).
[10] Hamed, A., & Colomb, R. M. (2011). End to end development engineering. *JSEA*, *4*, 195-216.
[11] Wagelaar, D., & Straeten, R. V. D. (2007). Platform ontologies for the model-driven architecture. *European Journal of Information Systems*, *16*, 362-373.
[12] Noy, N. F., & Musen, M. A. (2000). Algorithm and tool for automated ontology merging and alignment. *Proceedings of the 17th National Conference on Artificial Intelligence*.
[13] Corcho, O., Fernández-López, M., & Gómez-Pérez, A. (2003). Methodologies, tools and languages for building ontologies. Where is their meeting point? *Data & Knowledge Engineering*, *46*, 41-64.
[14] Kremen, P., Smid, M., & Kouba, Z. (2011). OWLDiff: A practical tool for comparison and merge of OWL ontologies. *Proceedings of 2011 22nd International Workshop on Database and Expert Systems Applications* (pp. 229-233).
[15] Flahive, A., Taniar, D., & Rahayu, W. (2013). Ontology as a service (OaaS): A case for sub-ontology merging on the cloud. *The Journal of Supercomputing*, *65*, 185-216.
[16] Dinu, V., & Nadkarni, P. (2007). Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *International Journal of Medical Informatics*, *76*, 769.
[17] Kiefer, T., & Nicola, M. M. (2012). Generating Structured Query Language/Extensible Markup Language (SQL/XML) Statements. Google Patents.
[18] Colomb, R. M. (2009). Metamodelling and Model-Driven Architecture. Faculty of Computer Science and Information Systems, University of Technology Malaysia.
[19] Elsawi, A. M., Sahibuldin, S., & Abdelhadi, A. (2013). Introducing the open source metamodel concept. *Journal of Theoretical & Applied Information Technology*, *57*.

Ahmed Mohammed Elsawi received the degree in computer science & information technology from Computer Man College for Computer Studies in 2000, his master degree in computer science as well in 2002. Currently, he is pursuing his PhD degree in the field of model driven architecture at Universiti Teknologi Malaysia, Kuala Lumpur, Malaysia.

Ahmed is experienced in business intelligence and enterprise architecture professional. He is a ITIL certified who worked in different mobile companies in Sudan, South Sudan and Afghanistan.

Shamsul Sahibuddin is a professor and the dean of the Advanced Informatics School (AIS) at Universiti Teknologi Malaysia (UTM) Kuala Lumpur, Malaysia. His research interests are software process, software quality, and software modeling.