2013

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

A UML BASED COMPARISON OF MODEL TRANSFORMATION TOOLS

Adalia C. Hiltebeitel

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2013

Approved by

Advisory Committee

Dr. Devon Simmonds	Dr. Bryan Reinicke

Dr. Ronald Vetter

Dr. Laurie J. Patterson, Chair

Abstract

A UML Based Comparison of Model Transformation Tools. Hiltebeitel, Adalia C. 2013. Capstone Paper, University of North Carolina Wilmington.

This Capstone provides an overview of Model Design Engineering (MDE) and identifies some transformation tools for the Model Driven Architecture (MDA) approach. This capstone report provides an overview of the main MDE concepts with a focus on Unified Modeling Language (UML). It provides an overview of various model transformation tools, and selects three from the group for further evaluation. The findings of this report confirm that MOLA and BOTL are still in the developmental stages and do not satisfy identified ideal requirements of a robust model transformation tool. The third tool, Rational Rhapsody Developer is fully developed and can assist a system architect in building scalable, software intensive, and detail rich software programs through the life cycle of software development.

Table of Contents

Tables

Figures

Chapter 1: Introduction

Enterprise software development is an elaborate process; it is a laborious, time intensive, detail rich, maintenance complex endeavor. According to Rahmouni (2011), the development and maintenance of web applications is a complex error prone process. The size and complexity of modern programs are factors that lead to this. Separation of concerns and abstraction offer some of the best techniques to combat problem size and complexity in software design. The use of software models has become a popular way to utilize the principles of abstraction and separation of concerns. Models are now part of an increasing number of engineering processes including software engineering. However, in software engineering models are primarily confined to a simple documentation role instead of being actively integrated into the engineering process.

Model Driven Architecture (MDA) is an Object Management Group (OMG®) initiative, that defines an approach to develop software based on modeling and automated mapping of models to implementation (Czarnecki and Helsen (2003)). Model transformation is a central operation for model handling. In the scope of model-driven engineering, it is assumed that model transformations, as any other model-based tool, can be modeled, which means that they have to be considered themselves as models.

According to Syriani (2009), the transformation of models is the center of present model-driven engineering. Models and transformations are first class entities in model-driven engineering. Model transformations transform source models into target models. Syriani (2009) further explains location determination in executing transformations is very important. Determining the location transformation implies that repeated executions lead to the same output. Code to model and model to code transformations are also model transformations since programs can be considered as models. Tracing transformation

enhance model transformation debugging. Traces may be shown automatically in the target or source model or the traceability information may be recorded in separate storages.

Abstract representations of models are referred to as metamodels. For the software system development process to be achievable, the designed models have to be operated and manipulated in such a way that you can generate or derive the source code and finally achieve a running system application. Model transformations facilitate the representation of the whole system by merging the different models of the system and organizing the models of the system into a number of abstraction levels. The top layer captures the computational independent system model while the next layer captures the platform independent model and the platform specific model is represented in the lowest layer. The main objective is to derive the code of the system through model transformations from the top abstract representation, through the intermediate layered models. Different model layers add successive design and implementation details until the source code is produced.

Background

Model transformation has many uses. To begin with, it provides dynamic semantics to models of the formalism. It also describes the operational semantics of formalism languages. Refactoring is used in evolving or optimizing the design of models. Exogenous model transformations are used to translate models from one formalism format to another. Tracing model transformation is essential in debugging. Model transformation has various applications in industrial projects such as in the mobile, airspace, military and automobile industry.

According to Tamura and Cleve (2010), the introduction of MDE/MDA/MDD (Model Design Engineering/Model Design Architecture/Model Driven Design) ideas in software systems development has led to the development of meta-modeling and model transformation. The evolution towards the consolidation of models as a consistent, foundational, unifying concept for software-intensive system development needs the realization of ideal sets of characteristics for specification of model transformation. Software systems development is based on model concepts. The characteristics captured and those that are hidden or omitted in models depend on the modeler's interest of focus in the system. In various systems, there can be many models, one for instance capturing the system's visual interface, another for the constraints in the system flow of control, another for the system's security in its operations and others for capturing nonfunctional aspects of the system including the system's scalability and safety.

Model Transformation Tools

There are a number of transformation tools available today. A tool allows a system architect to manipulate assets in a development environment to create and implement transformations in the process of designing and implementing a software project. It allows a stakeholder to observe an aspect of a system or environment to determine if it meets their needs in terms they will understand. This project looks at several tools.

MDA enables the development of a large scale project based on open standards and hardware capabilities. Using tools models may be kept though out the development lifecycle while models are being fine-tuned from a high abstraction level, and ending at system implementation. As a result, use of tools enhance the organizational capacity since MDA and use of tools means the organizational capacity depends less on any

individual programmer and more on the tools used. Code strategies no longer have to be in the programmers head or in various documents, system details can now be captured succinctly and permanently in model profiles. This project examines several tools.

Project Objectives:

i. To identify various UML based model transformation tools that have clear attributes.
ii. To investigate and analyze several UML based model transformation tools
iii. To select 3 UML based model transformation tools and elaborate on their functionalities
iv. To create a table comparison of all tools that shows their functionalities and differences.

The guideline implementing these objectives was as follows: During the background literature review several transformation tools were evaluated. The tools address past challenges in model transformation and the research paper analyzes how they have been developed and how they operate. This allows for a greater comparison between transformation tools (those in the previous studies and in this one) and may be of more interest by the community.

Three transformation tools were chosen for comparison and evaluation. One of the transformation tools is the IBM product Rational Rhapsody Developer. Because this is a commercial product, it may be more in line with a tool of interest to a larger community. Two additional tools BOTL and MOLA were also reviewed. The tools used as the primary focus in this investigation are selected based on ease of use as determined by setup time and learning curve. Using these criteria, candidates were selected based on accessibility for installation, short configuration time, or utilized languages that were

common knowledge or quick to grasp. This selection process also served as a guide for future work in this field.

This research illustrates how the various tools function to perform model transformations. It analyzes the quality of the transformation against ideal tool requirements, and then compares the tools based on this and their features and modes of operation. Appropriate documentation of the classes was utilized in this research paper.

Transformations were observed UML-UML in the noted tools. Qualities observed and compared in the noted transformation tools, included popularity in usage, graphical notations, lexical notation, declarative nature, bidirectional, inheritance in terms of updating and maintenance of transformations, XML support, Text-to-UML (reverse engineering), UML-to-UML, UML-to-text which is needed to implement running system as well as for documentation purposes, UML tool independence, which is desirable so that the solution is not tied to a single UML tool. Other tool requirements such as absence of proprietary intermediate structures which increase the complexity for the transformation architect, traceability, metamodel-based that is the source and target metamodels which are explicitly defined and exploited in order to drive the transformation specification.

Microsoft Project was used to keep track of project tasks and deadlines. This aided in the scheduling of tasks involved in the evaluation as well as kept track of changes in the research project and schedule that occurred. A full view of project tasks and delivery dates can be viewed in Appendix A – MS Project Timeline.

A glossary of terms was appended to this study and can be viewed in Appendix B – Glossary.

Report Outline

The rest of the paper is organized as follows: Chapter 2 introduces MDE as a software development methodology which focuses on the manipulation and handling of domain models. Chapter 3 describes several UML tools with distinct attributes that are compared for key functionality differences. Chapter 4 presents the results of three (3) chosen tools, their operational procedures and a detail walk through of the transformation outcomes of the Rational Rhapsody Developer tool. Chapter 5 presents a discussion of the benefits of UML, and it discusses the analysis of the tools in the comparative approach to evaluate their operations. Chapter 6 expands on the outcomes on the project and suggests future research which can be conducted to improve upon UML based model transformation.

Chapter 2: Review of Literature Review and Analysis

UML based model transformation is one approach that can be used in model transformation. The tools considered have attempted to address some past issues to enable a comparison basis between the tools.

According to Grønmo (2001), more recent proprietary languages have a higher learning curve and need more effort to apply. Absence of proprietary intermediate structures increases the complexity for the transformation process. Tools that are metamodel based are increasingly being used.

Model-Driven Engineering aims to consider models as first class entities and also considers that the different kinds of handled items can be represented as models. MDE aims to provide model designers and developers with a set of operations dedicated to the manipulation of models and thus model transformation is a central operation for model handling. UML based model transformation tools support UML-UML transformation and are popular in use. They use graphical notations, lexical notation and are declarative in nature. UML tool independence is desirable so that the solution is not tied to a single UML tool.

Jouault, Allilaire, Bezivin, and Kurtev (2008) shows model transformations are very vital in Model Driven Engineering. Important MDE techniques and tools assists software engineers to perform tasks just as they are assisted by debuggers, compilers and classical integrated development environments in programming. Particular domain languages help in solving common model transformation tasks.

According to Tamula and Cleve (2010), model transformation tools should be independently implementable, specify internationalization degrees and handle security issues. Transformation tools should be declarative so as to support incremental execution

of transformation in such a way that the changes in source models can be transformed to changes in target models immediately. The abstract transformation syntax, QVT definition languages is defined as MOF metamodels. The transformation definitions describe the relationships between source MOF metamodels which are used to generate target metamodels. The source and target metamodels are similar. The optional requirements of transformation requires the support for transactional definitions to specify the parts of transformation definitions identified as suitable for commit and rollback in execution, the traceability of the transformation executions, the mechanisms for extension and reuse of generic transformation definitions and bi-directionality of the transformation definitions.

The major recommendations in QVT Request for Proposal include the support for reuse of transformation definitions, composition, and packaging and to build large transformation systems in a maintainable and organized manner, declarative constructions for queries, simple declarative specification and visual, graphical languages as well as support for hybrid language in transformation definitions.

Grønmo, Belaunde, Aagedal, Engel, Faugere, and Solheim (2005) assert QVT Merge has a transformation signature that supports multiple target models. Object orientation is supported via inheritance through the extension mechanisms of inherits, merges and extends. It also supports multiple source models through a transformation signature that allows any number of input parameters. It also enhances bi-directionality through graphic notation and textual relations that enable bi-directionality. It enhances support for reusability and modularity and handles incomplete transformations with complete pattern parameters through QVT Merge / Relations and QVT Merge / mappings. Source models can also be updated and it allows black-box interoperability

that is transformation modules, mapping rules and query operations can be declared without body functions. All transformations can be defined with the mappings part in a textual notation. Unidirectionality is supported by the graphical notation and language in text. Traceability is enhanced by four resolve operations that can trace from source objects to target objects or vice-versa. It has easy usage in complex transformations as well as simple transformations. QVT Merge has the advantages of good program code structure into separate manageable transformation rules and constraints, black-box integration and modularity. It is open and flexible which allows writers to select appropriate paradigms for transformation problems.

Model-Driven Engineering (MDE)

Model-driven engineering is a software development methodology which focuses on making and exploiting domain models. It makes the process of design easy through models of recurring design patterns in the application domain. It increases productivity by maximizing compatibility between systems, through reuse of standardized models. It enhances communication between people working on the system through standardization of terminology and the best practices in the application domain.

According to Abmann, Zschaler, and Wagner (2005), a model is a representation of reality intended for some definite purpose. It is an explicit external representation of a part of reality as seen by people who can use that model to control, manage, change and understand that part of reality. They represent the reality and have a causal connection to the modeled part of reality giving true representations in such a way that queries of the model give truthful statements about the reality or the manipulation of the model lead to reliable adaptions to the reality. Models can represent various kinds of reality such as languages or domains or systems in particular. They describe and specify systems and the

system environment for specific purposes or describe the behavior or structure. Model Driven Engineering is a type of refinement-based software development where models are connected systematically. Models must be connected in such way that the model elements are traceable from a more abstract model to a more concrete model or vice versa. This is attained through meta-modeling whereby metamodels define a set of valid models thus enhancing their exchange, serialization and transformation. Metamodels make statements of what can be expressed in valid models of certain modeling languages. They are prescriptive models of a modeling language. Language concepts and constructs in metamodels are captured by a meta-class. Metametamodels specify and represent metamodels. They specify languages forming language specification languages (meta-languages).

According to Jouault et al. (2008), in model driven engineering, models are the main artifacts used in development and model transformations are very important operations applied to models. Various specialized languages have been developed to specify model transformations. High quality support for tools is very vital for the industrial adoption and success of model driven engineering.

The use of models in software engineering is common. Model paradigms for Model Driven Engineering are effective if their models are sensible to the user's point of view in that domain. Models are developed with a lot of communication between the application domain users and the developers, designers and product managers of the application domain.

Models enable the development of systems and software, including executable actions, or the models can be built to a certain level of detail after which the code is written by hand separately. However, complete models are built including executable

actions. The code can be generated from the models ranging from system skeletons to complete deployable products. Model Driven Engineering refers to the various approaches of development based on the use of software modeling as a basic method of expression.

Koch (2005) asserts that MDE approaches are aimed at reducing the problems involved in providing techniques for the construction of models and the specification of transformation rules as well as tool support and automatic generation of code and documentation.

MDE involves building models independent of platforms, transforming them to technological-dependent models and to achieve automatic generation of codes which is based on the rules of transformation. There are two types of transformations – model to model transformations and model to code transformations. In MDE, models are actively incorporated into the engineering process whereby they are considered as first class entities. Many items used in software engineering such as the repositories and tools used can be represented and viewed as models.

MDE uses a model-driven approach to provide developers and model designers with a set of operations aimed at the manipulation of models. Advanced MDE types have allowed the industry standards which enable consistent results in applications. The continuous development of MDE has increased focus on automation and architecture.

Model transformation is a central operation in model handling. It is aimed at specifying ways of producing target models from a given set of source models. Model transformations are themselves considered as models in the scope of model-driven engineering because they can be modeled just like other model-based tools. Models form the basic pieces of model-driven architecture. Models are defined according to the

semantics of a metamodel. Models that respect the semantics defined by a metamodel are said to conform to the metamodel.

Grangel, Bigand, and Bourey (2010), show that MDE improves the process of software development significantly. You can link models at various levels of abstraction in classical forward engineering processes so as to produce code or produce models from code using bottom-up approach in reverse engineering. Model driven engineering uses models to facilitate the process of development. Transformations are the main operations on models used in information mapping between models. Approaches which are model driven are aimed at providing solutions so as to facilitate the processes of software development.

Unified Modeling Language (UML)

Mitchell (2003) asserts that UML is a language that is used in documenting software system artifacts. UML is used to construct, visualize and specify the elements of software systems. It uses graphical notation to model software system aspects. UML is a standardized general-purpose modeling language in object-oriented software engineering which uses graphic notation techniques to create visual models of object-oriented software-intensive system. It is method-independent and extensible and is used to specify, construct, visualize, modify and document the artifacts of an object-oriented software-intensive system under development.

UML uses diagrams to give a person a mental visual model of the real system. Examples of diagrams used in UML are use case scenario diagrams, activity diagrams and class diagrams. According to United Modeling Language (UML) Basics (n.d.), UML is a standard for technical exchange of designs and models. It is not a programming language and does not indicate a particular process like model transformation tools,

rather, it is a standard language that enhances the specification, visualization, constructing and documenting software system products.

A model is a pattern that indicates how something in the real world can be constructed. They represent visible details and ignore other details. They are used in UML to simulate the understanding of the represented objects, are easier to build and change as one learns the problem or task at hand. Use case diagrams in UML enhance communication between the developers, stakeholders as well as the developers in the software project planning. They model the system environment while showing external actor's connections to the system functionality.

Class diagrams are used in UML to show the operations and attributes in a class and the constraints for the way objects collaborate. They describe the different object types and the static relationships between them in the system. They show the static structure of the domain and system abstractions.

Package diagrams show the breakdown of the large systems into logical groupings of small subsystems. They show the groupings of classes and dependencies and between them.

Activity diagrams show the sequential flow of activities and complement class diagrams by showing business workflow. They enhance the discovery of parallel processes which prevents unnecessary sequences of the business processes.

State transitions show the possible states that class objects can possess and the events that cause their changes. Sequence diagrams show the dynamic collaboration between objects for sequences of messages send between them in time sequences. They are used only when operation sequences have to be shown.

Collaboration diagrams show actual objects and links between them as a network of objects. Component diagrams show the software components which can be dynamically linked libraries, executable codes, binary codes or source codes. Deployment diagrams show the physical system software and hardware architecture. They highlight the physical relationship between the system hardware and software components. Diagram components represent physical code modules and correspond accurately to package diagrams.

Amelunxen and Schurr (2008) show that UML provides an intuitive and easy language for many domains. It is influenced by the experiences from other modeling approaches. According to the Rational e-development company (2000), the importance of UML is broad. It is used in object oriented software development as well as other software development areas such as in facilitating the ability of practitioners to communicate their needs and assess their teams and in the modeling of data. UML is used to elaborate the development of object relational databases from business requirements in the physical model of data. UML supports the needs of modeling data and facilitates the development of software and data modeling.

According to Bell (2003), UML was created as a unifying language to enable professionals in information technology to model computer applications. It is not a methodology and does not need formal work products rather it provides diagrams that can be used in different methodologies to improve understanding of software applications under development.

UML diagrams just provide good introduction to languages and principles behind its use. Exforsys Inc (2007) shows that UML enables developers to develop high quality applications. Designed around the object oriented approach, it is robust and flexible and

provides developers with high communication levels. It can work with many operating systems and hardware and function with many programming languages notably Java.

UML tools also allow us to study the source code and to reverse engineer to allow it to be seen as UML diagrams. There are many UML tools that enable the execution of UML in various ways. Some UML tools execute the model in such a way that they allow the user to define his needs. Some UML tools are designed to work with specific domains and can create the language code using UML which produces deployable applications that are bug free. Code generators with the right scalable pattern run quickly.

UML is a specification language in the software engineering field and it is model driven. It uses a powerful palette independent of middleware applications. It gives a choice of making an independent or specific platform. It is a general purpose language that uses graphical designation to create abstract models which are then used in systems.

UML models can be transformed into other forms. This is achieved through the use of transformation tools similar to QVT. System diagrams are a portion of the UML graphical symbol used to denote the system's model. Each model has particular role to ensure proper functioning of the system. A model can be exchanged among UML tools through the proper file format.

A Taxonomy of Model Transformation

Taxonomy is a way of naming and organizing things into groups which share similar qualities. It can aid a software developer to select a particular model transformation approach that is best suited for his needs and help tool builders to assess the strengths and weaknesses of their tools compared to other tools or help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

An example of multiple source models is model merging, where the aim is to combine multiple source models that have been developed in parallel into one resulting target model. An example of multiple target models is a transformation that takes a platform-independent model (PIM), and transforms it into a number of platform-specific models (PSM). Both examples are schematically represented in Figure 1.

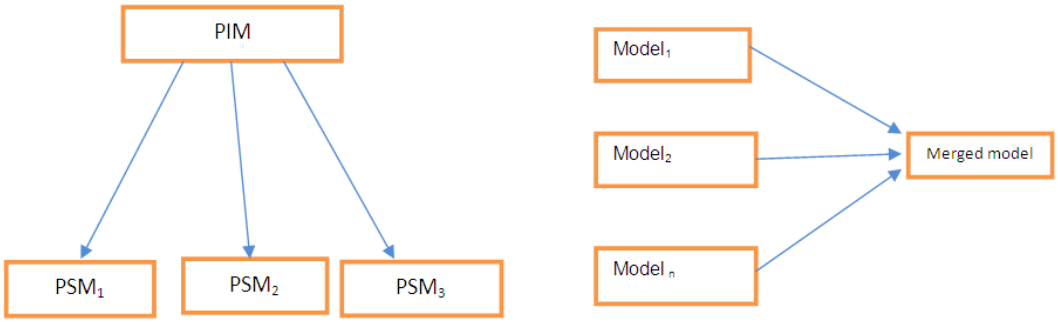


Figure 1: Examples of model transformations. From: Mens, T, Gorp, PV 2006, A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125–142.

Mens, Czarnecki, and Gorp (n.d.) show that the kind of artifacts being transformed is important in evaluating the transformation approach. Exogenous transformations are transformations between models using different languages while endogenous transformation involves models expressed in the same language. Exogenous transformation can involve migration from a program written in one language to another while maintain the same abstraction level, reverse engineering, is the opposite of synthesis and involves the extraction of a high level-specification from a lower one and the synthesis of a higher level, more abstract specification into a more concrete one.

Vertical transformations are conducted where the source and target models are located at different abstraction levels while horizontal transformations are transformations where both the source and target models are in the same level of abstraction. Another distinction is whether the source and target models belong to the

same technological space. The QVT specification standardizes a language for writing programs in model transformations of different technological spaces.

According to Tamula and Cleve (2010), classification of model transformation tools is based on existing variability and commonalities. The classification of model transformation tools is based on whether they are operational or declarative, multiplicity or directionality, the schemes of re-use such as decomposition and composition of transformations, groupings, higher order or generic re-use schemes. It is also based on the preservation that is the correctness, behavior and structure, the technological space of the target and source domains, the levels of abstraction which can be vertical or horizontal and whether they are exogenous or endogenous that is whether they have the same or different domain metamodels.

The author further states classification of model transformation are based on their functional requirements including the support for the change propagation, traceability, specific bidirectional transformations, specific high order and generic transformations, the ability to verify, validate and test transformations as well as deal with inconsistent or incomplete models, the guarantee of correctness or reuse of transformations, the applicability criteria that is the conditions applied to particular transformation sets, the ability to create, delete, update and read transformations as well as the customization or reuse of transformations.

Non-functional requirements include the standardization and conformance to other standards, scalability, verbosity and conciseness and the usability and usefulness in enhancing practical purposes, intuitive in use efficiency. Important characteristics of model transformation tools include formal mathematical and proper foundation which facilitates verification, enforcement and preservation of properties including semantic and

syntactic correctness in the definition of transformation and the automation of the process of transformation, verifiability and testability, mechanisms for scalability and reuse in inheritance of transformations which include higher order, generic decomposition, composition and groupings of transformations as well as automation in the transformation tool implementation / execution.

According to Tamula and Cleve (2010), top level characteristics of model transformation tools are the source-target model relationship; rule application control and organization in re-use mechanisms, modularization and packaging, the application scheduling and location. Another top level characteristic is the rules of transformation in domain languages, aspects, reflection, parameterization, intermediate structures, application conditions, form and structure and the specification of pre and post conditions both executable and non-executable.

The sub-categories of model transformations include hybrid transformation tools, graph transformation, relational, template based, operational based, structure driven and direct manipulation.

Hybrid transformation combines two or more approaches while graph transformation is based on graph rewriting over extensions and variations of types of attributed labeled graphs. The relational approach groups declarative approaches based on mathematical relations while template based grouping uses model template specifications. The specifications of model templates are normally expressed in concrete syntax of the target language which represents the transformation results. Structure driven groupings are based on transformation frameworks while operational groups are based on direct manipulation with more transformation support. Direct manipulation is based on some internal model presentations and operational sets for model manipulation.

The main attributes of the QVT specification are pattern matching via object template expressions in relations and automatic management of traceability in transformations. It is incremental and multi-directional and adopts OCL which is a purely declarative language for queries. It has a hybrid model to model transformation approach and it has a MOF-meta-modeled abstract syntax with concrete graphical and textual syntaxes.

The hybrid model transformation language of the QVT specification follows a combined fine-grained and coarse grained approach. The relations and core parts of the declarative portion of QVT define execution semantics in the imperative part.

The QVT specification provides support for two mechanisms in the use of imperative transformation styles that is the non-standard Black-Box MOF operational implementations and the standard operational mappings language. Each relation in QVT defines a class to be instantiated to trace between model elements being transformed and which had a one to one mapping to an operation signature implementation.

ATL (ATLAS Transformation Language) Transformation Tool

ATL is a model transformation language and tool. In MDE, it provides ways to produce a set of target models from a set of source models. It can be used to do syntactic or semantic translation and is built on top of a model transformation Virtual Machine. Rahmouni (2011) asserts that ATL is a domain specific language which is used in specifying model to model transformations. It was developed in the framework of the ATLAS project for model to model transformation. It is a model transformation language that enables the specification of how one or more target models can be produced from various source models. It introduces concepts that enable model transformations.

Jouault et al. (2008) asserts that ATL is a hybrid language. Hybrid languages provide a mix of imperative and declarative constructs. It is a model transformation language whose execution environment is based on the Eclipse framework. ATL is a language of model transformation which is specified as a concrete syntax and metamodel and it supports major tasks involved when using a language such as debugging, executing, compiling and editing. The declarative style of transformation is mostly used because simple mappings can easily be expressed. However, imperative constructs are provided to enhance complex mappings which are difficult to handle declaratively. ATL transformation programs consist of rules which define how source model elements are transformed in order to create and initialize target model elements. Model transformation oriented virtual machines are defined and implemented to enhance the execution for ATL while maintaining certain levels of flexibility. However, specific transformations from the ATL metamodel to the virtual machine must exist.

Extending ATL involves specifying the new language features execution semantics in the form of simple instructions such as basic actions on the models like property assignments and element creations. ATL transformations are unidirectional, operating on read- only source models and producing write-only target models. While a transformation is being executed, the source model may be navigated but changes to it are not permitted. The target models cannot be navigated.

A bidirectional transformation is implemented as a series of transformations, one for every direction. Transformation definitions in ATL form modules. Modules have a mandatory header section, import section, and a number of helpers and transformation rules. The header section contains the name of the transformation module and declares the target and source models. It contains the key word module which is followed by the

module name after which the source and target models are declared as variables typed by their metamodels. A helper originates from OCL Specification. There are two kinds of helpers: the attribute helpers and the operation helpers. In ATL, helpers can only be specified on an OCL type or on a source metamodel type because target models are not navigatable. Operation helpers elaborate operations in the model element context and can have input parameters and use recursion. They are used to associate read-only named values to source model elements. Attribute helpers can refine source models before the transformation is executed.

Transformation rules are constructs in ATL which are used in expressing the logic of transformation. Rules in ATL can be specified imperatively or declaratively. Declarative ATL rules comprise of target and source patterns. The source pattern of the rules specify source type sets which come from the set of collection types and from the source metamodel and a guard which is an OCL Boolean expression. Source patterns are evaluated to sets of matches in source models. Target patterns are composed of a set of elements where each element specifies the target types from the target metamodel and sets of bindings. Bindings refer to features of the target types that are the association end, a reference and an attribute. They specify initialization expressions for feature values.

Several types of matched rules differ in the way they are prompted. Unique lazy rules are triggered by other rules and are applied once for a particular match. When a unique lazy rule is triggered later on the same match the already created target elements are used. These types of rules are indicated using unique lazy key words. Lazy rules are triggered by other rules. They are applied on one match severally as it is referred to by other rules, each time producing new sets of target elements. They are indicated by the key word “lazy.” Standard rules are applied once for each match found in the source

models. An ATL resolution algorithm takes care of triggering lazy rules and unique lazy rules when source elements are referred to in initialization expressions. Rule inheritance in ATL is used to specify polymorphic rules and to reuse codes. The execution begins by invoking an optional called rule which is marked as an entry point. This rule can invoke other called rules. The algorithm executes the standard match rules but rule matching and application are separated in two phases. The ATL engine handles the main tasks of ATL such as compilation and execution. ATL transformations are compiled to programs in specialized byte-code which is executed by the ATL virtual machine.

Stephan and Stevenson (n.d.) show that ATL is strongly typed and supports built-in collection modifiers. It is a well-rounded transformation tool with an easy to use declarative syntax and a more expressive imperative syntax. It attains modularity through using helper functions and transformation staging. ATL is simple but powerful. It also strikes a good balance between ease of expressiveness and use.

Rational Rhapsody Developer

According to the IBM Corporation (2009), the IBM Rational Rhapsody developer development provides a model-driven development environment that enhances the process of software automation. It is UML based. It uses graphical models to generate software applications. This facilitates the reuse of assets, enhances team collaboration and helps in the identification of design errors and defects in software development. It enables static checking of models, model verification, execution and simulation with automatic creation of diagrams, domain-specific language support form modeling and analysis, external code integration and supports functional object oriented, object-based and functional paradigms. It enables the improvement of code quality due to

comprehensive modeling requirements and traceability capabilities. With application generation capability, it boosts productivity.

Rational Rhapsody provides a visual design environment for creating model systems and requirements in the unified modeling language diagrams. It enhances testing through the simulation of applications on local hosts or remote targets to debug within simulated views and enhances implementation through automatic code generation from analysis models, design through tracing design requirements while taking into account detailed considerations of the design as well as mechanistic and architectural considerations.

Rational Rhapsody also enables analysis through the derivation of the systems behavior, structure, the system architecture and requirements. Rational Rhapsody generates complete application code form flow charts, activity diagrams, state charts and class diagrams which are usually easy to debug and read. It supports UML 2.1 and thus its users can get useful extensions such as structured classes and ports.

Yet Another Transformation Language (YATL) Transformation Tool

YATL is a hybrid language that is a mixture of declarative and imperative constructions designed to express model transformation and responds to QVT Request For Proposals by the OMG.

It is described by an abstract syntax and a textual concrete syntax but as of this writing does not yet provide a graphical concrete syntax. According to Patrascoiu (2003), models in YATL are expressed as a set of transformation rules. YATL supports the requirements of QVT. It provides all the required computational power regardless of the language or platform. Transformations, views and queries are grouped in namespaces so as to provide reusability and avoid collision. YATL has well defined semantics with

deterministic transformation rule transformation. Its transformation engine can perform transformation efficiently even for large scale systems. Quartel et al. (n.d.) show that a defined YATL transformation for a composite concept Messaging which transforms a source model into a target model such that each instance of Messaging in the source model is replaced by its corresponding composition of elementary concept instances in the target model. The rule messaging defines the creation of the constituent concept instances and associations between them. The use of the UML profiling mechanism facilitates the efficiency of general purpose languages with the intuitive clarity and ease of use of dedicated languages. An extension of UML's Profile package supports the specification of the composite stereotypes.

MOLA Tool Architecture

Kalnins and Clems (n.d.) describes MOLA Tool implementation of the MOLA transformation language. It consists of two parts - the Transformation Development Environment (TDE) and the Transformation Execution Environment (TEE).Execution of MOLA transformations are performed using MOLA Transformation Execution Environment (MOLA TEE).The most important for wide usage of MOLA is the compiler version generating Java against the API of Eclipse EMF, which is the most popular model repository kind so far. The result of compilation is the executable file (jar or dll). The definition of corresponding repository is also created by MOLA compiler. Figure 2 provides an example of MOLA tool compilation.

The MOLA program gives complete descriptions for formal mapping between two notations which are at times complex – involving several context conditions and classes and thus cannot be specified using pure descriptions (Kalnins and Vitolins, n.d.).

Kalnins, Celms, and Sostaks, (2004) argue that mappings in MOLA are dynamically defined by model transformations.

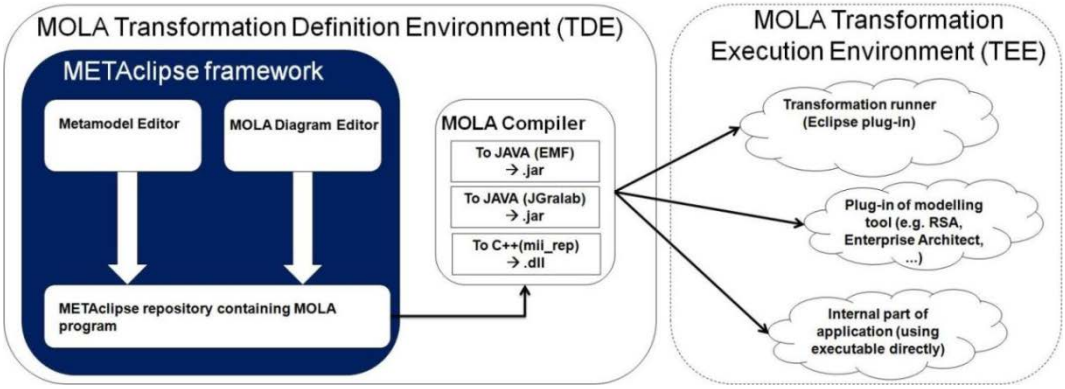


Figure 2: MOLA tool compilation. From: MOLA Project 2011, Mola tool architecture, viewed June 1, 2012, http://mola.mii.lu.lv /tool_description.html.

XSLT Transformation Tool

According to Clark (1999), the XSLT Mapper enables creation of data transformations between source scheme elements and target scheme elements. The XSLT Mapper transformation tool is used to create the contents of a map file. Figure 3 shows the layout of the XSLT Mapper.

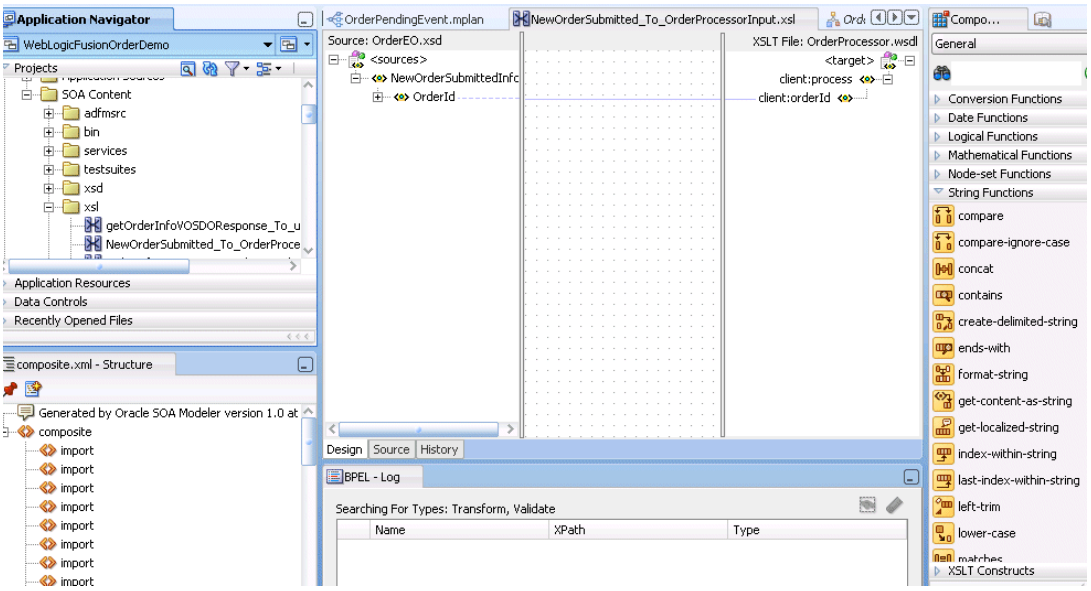


Figure 3: Layout of the XSLT Mapper. From: Oracle 2011, Oracle Fusion Middleware Developers Guide for Oracle SOA Suite, viewed June 1, 2012, http://docs.oracle.com/cd/E21764_01/integration.1111/e10224/bp_xslt_mpr.htm.

The Source and the Target schemes are represented as trees. The nodes in the trees are represented using a variety of icons. According to Clark (1999), the displayed icon reflects the scheme or property of the node. Transforming XML with XSLT (n.d.) asserts that XSLT is used to transform database-drive XML into HTML pages, emails, SQL scripts, and XML datagrams of particular vocabularies. XSLT is a W3C standard language for describing transformations between XML documents. It describes rules which are used to transform the source XML into a result XML. The XSLT processor carries out the transformation based on the transformation rules. XSLT style sheet uses XSLT vocabulary to describe transformations.

UMLX Transformation Tool

The transformation comprises an annotated copy of the scheme for the user models together with a model for each of the scanning and building phases which are used by the three major transformations Annotate Usage, Scan and Build. The intermediate models in the compiled transformation are the result of compilation at of the transforms and scheme. According to Willink (2003),

> "*each of the incoming scheme and transforms is annotated with properties that assist in the production of the scan and build models by the Generate Scan and Create Build transforms identifies which composition arc in the schema is instantiated by each composition arc in a transform*."

The increased use of modeling techniques has led to the urge to use models as a programming language as a part of MDE. This urge has been fulfilled by using XMI for model interchange (A model can be exchanged among UML tools through the XMI file format), and XSLT for model transformation. This has led to the development of the

UMLX which is a graphical transformation tool with minor extensions to UML; it is a high level tool for transformations. It uses standard UML class diagrams to define information schemas and their instances and extends class diagrams to define inter-schema transformations.

Bidirectional Object Oriented Transformation Language (BOTL) and Tool

BOTL is mathematically founded. It allows reasoning about properties of transformations and to specify transformations among object oriented models. It allows verification of the desired properties of applicability, and metamodel conformance at specification time. It allows for specification mappings between the different model layers of the MDA. It can also be easily extended to specify transformations on a single model.

As shown by Marschall (2004), presently XSL transformations are used to transform XMI (OMG-XMI, 2002) representations of models. XSL transformations [XSL99] are widespread as a language for the specification of transformations of XML documents into other textual representations. Thus XSL transformations are often used for the integration of applications that exchange different kinds of XML documents. XSL documents lack an intuitive, graphical representation and have some more disadvantages. They operate on XMI representations and thus writing MDA transformations, according to Wagner and Giese, are said to be a long winded and error-risk task; which led to the introduction of BOTL for the transformation of object oriented models. BOTL offers a UML-like notation to specify rules which is comparable to graph transformations. BOTL is not incremental in its approach but rather it is batch-oriented, Wagner and Giese (2009).

Wang (2005) asserts that the ArgoUML4BOTL is an editor for BOTL rules and metamodels. It generates BOTL-XML specifications. BOTL is founded on the formalization of UML class diagrams. It serves as a mechanism for specifying the integration of development models and the description of tool chains. It enables one to verify whether the specifications are bijective, whether they will produce models that conform to particular metamodels and whether a BOTL specification is applicable. It is founded on formal, comprehensive graphical notation. BOTL transformation rules are specified using simple UML-like notations. The prototype model transformation tool is based on Argo UML.

Visual Model Transformation (VMT) Tool

The transformation language is a visual declarative language that supports the specification, composition and reuse of model transformation rules. These rules, as shown by Oldevik (2003), make use of the OCL language and a visual notation to indicate the selection, creation, modification and removal of model elements. According to Taentzer (n.d.), the visual model transformation tool was developed to enhance visual modeling.

Sendall, Perrouin, Guelfi, and Birberstein (n.d.) show that VMT supports the re-use, composition and specification of model transformation rules in a visually declarative manner. VMT rules use the OCL language and a visual notation to represent the removal, modification, creation and selection of model elements. The execution process of a transformation involves the application of one or more transformation rules to the source models according to the flow control as defined by the rule ordering schema. VMT provides a schema of rule ordering that is used by users to formulate graph

transformation in a chain of simpler transformations. Transformations are defined by specifying transformation rules in the order in which they are to be executed.

Chapter 3: Methodology

The following transformation tools are briefly reviewed and discussed: ATL (ATLAS Transformation Language) transformation tool, Bidirectional Object oriented Transformation Language (BOTL) transformation tool, XSLT transformation tool, YET ANOTHER TRANSFORMATION LANGUANGE (YATL) transformation tool, UMLX transformation tool, VISUAL MODEL TRANSFORMATION (VMT) transformation tool, MOLA TOOL ARCHITECTURE transformation tool, and the Rational Rhapsody transformation tool. These tools are all UML based and have attributes that can be identified. These tools address past challenges in model transformation and it was analyzed how they have been developed and how they operate. This research also illustrate how these tools function to perform model transformations and then analyze the qualities of transformation and the tool requirement and then compare them based on this and their features and mode of operation. Table 1, on the following page, shows a comparison of the transformation tools.

According to Mens, Czarnecki, and Gorp (n.d.), the quality requirements for a transformation tool include the standardization of the transformation tool. A transformation tool should be compliant to all the relevant standards. Transformation tools should also be acceptable to the user community. It should also have mathematical properties to prove its theoretical properties such as its correctness, completeness and termination. It should also be scalable that is cope with large, complex transformations or transformations of large, complex models. Model transformation tools should be concise that is have as few syntactic constructs as possible. It should be useful in serving a specific purpose relating to the training of developers and their experience. The

taxonomy of model transformations enables developers in choosing a specific transformation tool in accordance to his particular requirements and needs.

Table 1

Comparison of Transformation Tools

	Transformation Tool							
Requirements	MOLA	BOTL	VMT	YATL	XSLT	ATL	UMLX	Rational Rhapsody
Commonly used language	No	No	Partly	No	Yes	No	No	Yes
Inheritance	No	No	Yes	No	No	Yes	No	Yes
Graphical notation	Yes	Yes	Yes	No	No	No	Yes	Yes
Lexical notation	No	No	Yes	Yes	Yes	Yes	No	Yes
Declarative	No	Yes	No	No	Yes	No	Yes	Yes
Bidirectional	No	Yes	No	No	No	No	No	Yes
XML support	No	No	No	No	Yes	No	No	Yes
Text-to-UML	Yes	No	No	No	Yes	No	No	Yes
UML-to-UML	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
UML-to-TEXT	No	No	Yes	No	Yes	Yes	No	Yes
UML tool independence	No	Yes	No	Yes	Yes	Yes	Yes	Yes
No proprietary intermediate structures	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Traceability	No	No	Yes	Yes	No	No	Yes	Yes
Metamodel/MOF-based	Yes	Yes	No	Yes	No	Yes	Yes	Yes

Table 1: A basic analysis of the tools discussed. From: Grønmo, R. (2001). An empirical study of the UML model transformation tool (UMT). SINTEF: Norway. Retrieved October 31, 2012, from http://heim.ifi.uio.no/~roygr/INTEROP-ESA-2005.pdf

As asserted by Wagelaar, Straeten, and Deridder (2009), model transformations are expressions in model transformation languages distinct from the execution of model transformation itself. The execution of model transformation refers to as the act of

transforming models. The properties of model transformation such as reusability, maintainability, and scalability are very essential and the composition of model transformation permits the creation of smaller, reusable and maintainable definitions of transformation that can be used together to perform larger transformations. The application of model transformation has increased in model-driven engineering with various model transformation tools and languages. Unlike in the past where the major focus of model transformation was the expressiveness of transformation languages, other properties such as the reusability of model transformation definitions, maintainability and scalability have become more important.

An Analysis of ATL

A summary of Jouault's (2008) explanation is that ATL is a model transformation language and tool that uses imperative systems and declarative rules. It makes a distinction between input models and output models. A model cannot function as an input and output model in one single transformation execution. Output models are write-only and always empty at the start of a transformation execution while input models are read only. ATL supports three unit types: modules, queries and libraries. ATL transformation modules have several input models but only one output model. They run one transformation module at a time. Rule-based model transformation languages support rule-level adaption like operational mappings and QVT relations. When an ATL transformation is launched, the base transformation module and the models are loaded first then each ATL library is loaded and its operations registered in the lookup table operation of the execution environment.

An Analysis of BOTL

Tratt (2008) argues that many model transformation approaches share the same property of being stateless meaning after an initial transformation, the only possible action is to run the transformation again, from the start creating an entirely new target model irrespective of the target model that has been created previously. BOTL and XMOF tackle this problem of change propagating transformation models.

Marschall and Braun (2003) assert that BOTL is precise, allows reasoning on the applicability of specifications, and enables us to verify that only valid models are created. BOTL supports the verification of the desired properties. A BOTL rule consists of a number of rules that create the model fragments. The order of the application of the different rules and the order of the pattern matches does not alter the result of a BOTL transformation.

For automatic transformation of models with tool support, such as the generation of a PIM from a PSM, it must be ensured that the transformation does not fail due to incomplete or inconsistent specifications. Model transformation languages must have notions of applicability and permit automatic reasoning about it. BOTL has this attribute. A user can write BOLT rules that define model transformations. A BOLT rule set is applicable if there are no two rules that generate two objects of the same type whereby the same attribute gets assigned a value twice or if of its rules alone is applicable. BOLT provides some primary algorithms to determine the source variables the value created from a particular attribute of a target object variable relies and the source object variables and identity created from a particular object variable depends. It also determines whether it holds for two sets A and B of source object variables that whenever the elements of X

match to the same source objects, then the elements of Y match always the same source objects too.

Applicability means that a model transformation specification is possible. Metamodel conformance means that a transformation will produce target models depending on the target metamodel. A model is regarded as a metamodel conform if every association in the model are of a type that occurs in the metamodel and connect objects of the correct types and if every object in the model are of a type that occurs in the metamodel. A model can also be regarded as a metamodel conform if every object in the model does not have less outgoing associations of a type than the class association permits. This is referred to as lower bounds conformance. It can also be regarded to as a metamodel conforms if every object in the model has no more outgoing associations of a type than the class association does permit. This is referred to as upper bounds conformance. These properties are guaranteed by the BOTL syntax rules. BOTL specifies mappings between different model layers of the model driven architecture.

BOTL facilitates reasoning around the bijectiveness of transformations and metamodel conformance where the property that the application rules sets can only generate target models conforming to given target metamodel. It also facilitates applicability. The aim of BOLT is to allow developers to express intuitive knowledge on how models relate easily. It has well defined semantics to allow the generation of model transformers. BOLT provides a specification mechanism to describe the integration of development models and tool chains. It is used to define the transformation of data between ASCET-SD, the UML suite and CASE tools DOORS. BOLT is based on a formalization of UML class diagrams. In BOTL formalism, the metamodel notation is

used to define data models used within tools. Figure 4 shows a sample BOTL metamodel represented as a UML class diagram.

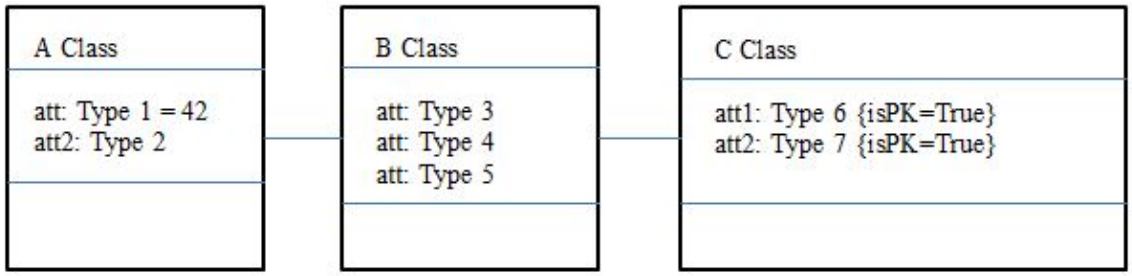


Figure 4: A sample BOTL metamodel represented as a UML class diagram. From: Marschall, F., and Braun, P., Model Transformations for the MDA with BOTL. (2003). Boltzmannstr: Institute for Information. Viewed October 27, 2012, http://www4.informatik.tu-muenchen.de/publ/papers/marschall_braun-mdafa03.pdf.

In the representation of BOTL model variables, UML profiles for object diagrams are used. Figure 5 shows an object variable regarding the metamodel in Figure 4.

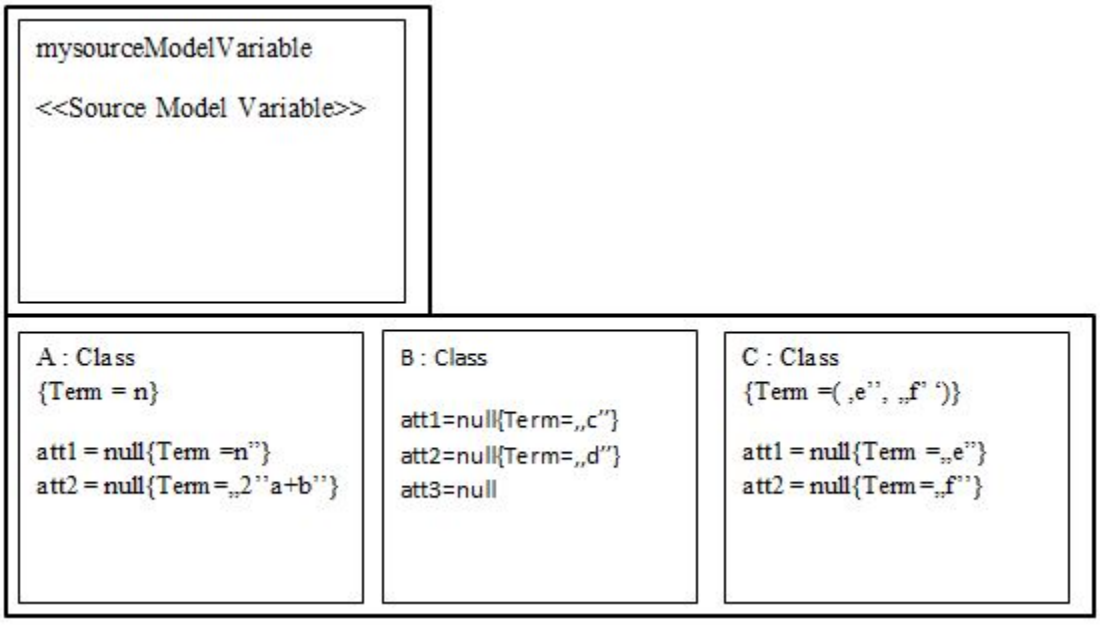


Figure 5: A sample BOTL model variable represented as an UML diagram. From: Marschall, F., and Braun, P., Model Transformations for the MDA with BOTL. (2003). Boltzmannstr: Institute for Information. Viewed October 27, 2012, http://www4.informatik.tu-muenchen.de/publ/papers/marschall_braun-mdafa03.pdf.

An Analysis of XSLT

Wagner and Giese (2009) argue that model-driven software development needs appropriate model transformations which can be applied in various software development stages. The transformations have to effect these changes to ensure proper model synchronization. The execution of transformation rules is essential in achieving incremental model transformation. The model transformation framework developed by IBM validates QVT concepts. It has the capability to synchronize node addition and removal. However, it cannot support defining custom constraints. XSLT does not support incremental change propagation. Visual transformation tools are based on theoretical work on graph grammars and graph transformations.

An Analysis of YATL

Garching and Octavian (2003) advise that a YATL program consists of one or more translation units. Additionally, each is contained in a separate source file. When a YATL program is processed, all of the translation units are processed together. Each translation unit is contained in a separate file. Thus, translation units can depend on each other, possibly in a circular fashion. Translation units consist of zero or more import directives followed by zero or more declarations of namespace members: queries and views. Namespace concept is used in YATL to prevent name collision. Translation units have zero or more import directives which are followed by zero or more declarations of namespace members that are transformations, views, and queries.

YATL queries are OCL expressions which are evaluated in given contexts such as operations, properties, classifiers and packages. YATL transformations maps the source model instances by matching the patterns in a source model instance and creating collections of objects with given properties in the target model instance. While the

creation part is done by the imperative features of OCL, the matching part is done by the declarative features of OCL. To support the mandatory requirements, a YATL transformation describes a mapping between a source MOF metamodel A, and a target MOF metamodel B. The transformation engine uses the mapping to generate a target model instance conforming to B from a source model instance conforming to A. The source and the target metamodels may be the same metamodel. Each transformation contains one or more transformation rules. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code. This approach allows filter expressions to include both modeling information (e.g. navigational expressions, properties values, collections, etc.) and platform dependent properties (e.g. special conversion functions), which makes them extremely powerful. A compound statement specifies the effect of the RHS. The LHS and RHS for the YATL transformation are described in the same syntactical construction, called transformation rule. A rule is invoked explicitly using its name and with parameters. The abstract syntax of YATL namespaces, translation units, queries, views, transformations, and transformations rules is described in Figure 6.

YATL declarative features originate from the description of the LHS of transformation rules and the OCL expressions. OCL describes the matching part of YATL rules since it is a well-defined language used in querying UML models. It provides a standard library in acceptable computational expressiveness. YATL supports various imperative features to enhance the operations of building statements to simplify the construction of target model instances, native statements to interact with host

machines, iteration, decision and declaration statements and the creation and deletion operations. The abstract syntax of YATL statements is shown in Figure 7.

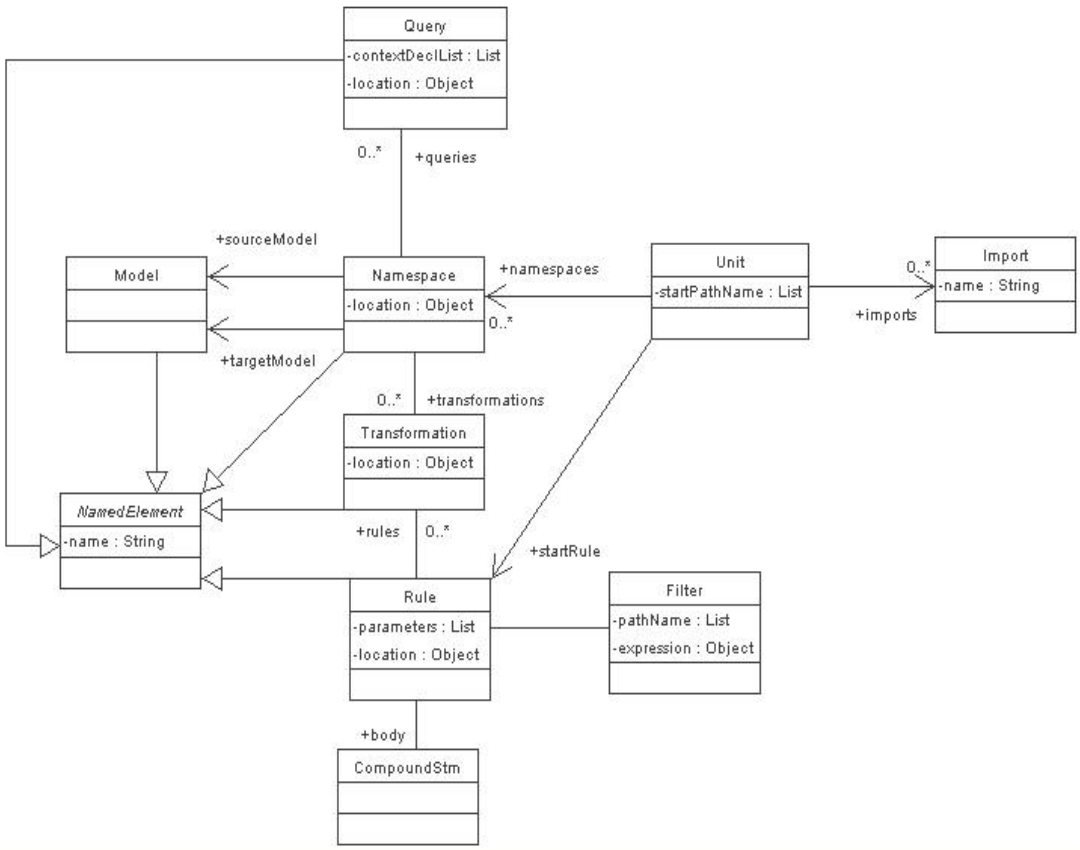


Figure 6: YATL Model Transformation. From: Patrascoiu, O. (n.d.). Yet Another Transformation Language. United Kingdom: University of Kent.

YATL supports explicit creation or destruction of objects imperatively. For traceability, YATL has a track operator that are defined using imagine, name and domain concepts which facilitates traceability in large scale systems.

An Analysis of UMLX

Willink (2003) surmises that in principle both XSLT and UMLX are very powerful tools for model transformations. The XSLT line count is perhaps 80% smaller than Java. This level of improvement is not often found between good programming languages. It arises because XSLT has language syntax tuned to the problem of

transformation, whereas Java support is restricted to classes rather than syntax. XPath expressions, which are used to navigate models, have become more powerful in XSLT 2.0. However they are a relatively independent sub-language for which it is difficult to provide compile-time validation of intent. As a result it is very easy to make mistakes in the terse syntax XSLT and XPath are therefore perhaps may be too powerful in terms of ease of use. Willink asserts like an assembly language, they allow all problems to be resolved but provide insufficient constraints to validate and steer programmers towards sensible solutions. A more restrictive environment is needed, and a language such as UMLX provides it. UMLX may be translated to XSLT, if XSLT is required, but may also be translated direct to C or Java code if that is preferred.

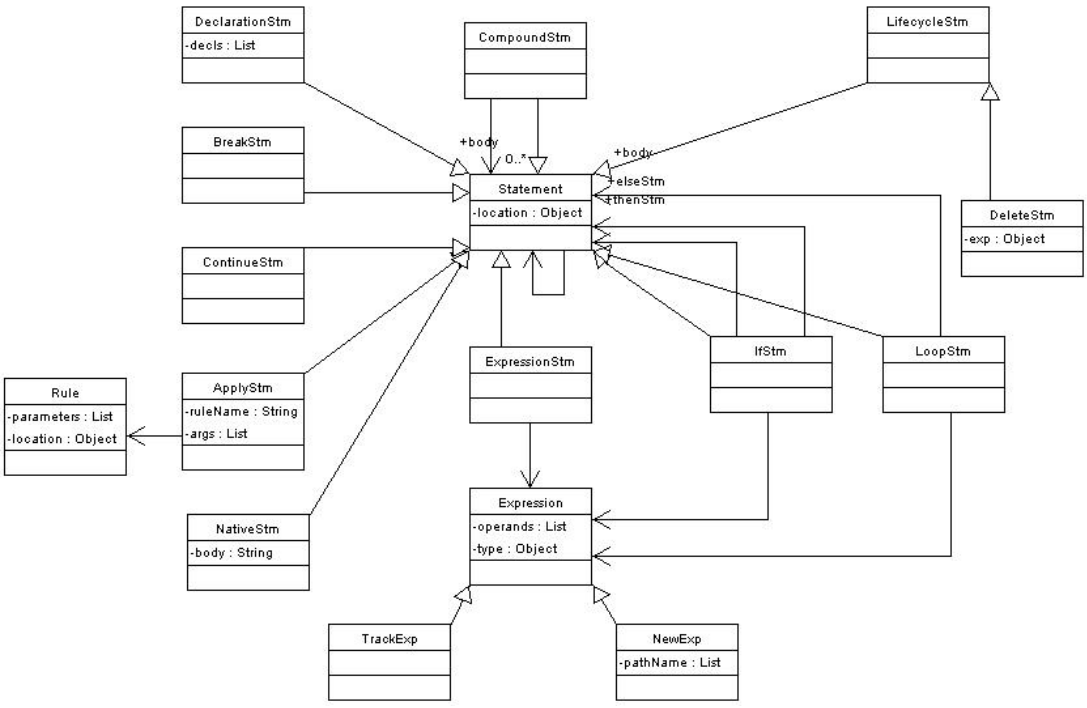


Figure 7: A Statement in YATL. From: Patrascoiu, O. (n.d.). Yet Another Transformation Language. United Kingdom: University of Kent.

UMLX, says Willink (2003), makes extensive use of schemas both for the user models and for the transformations. As a result UMLX can clearly specify what needs to be done. This provides the opportunity for optimizations that would be hard to perform in XSLT where there is much less understanding of the problem, and in many cases an explicit definition of an implementation. Willink has identified that XSLT can give an 80% line count saving over Java, so how does UMLX compare to XSLT? At present only small examples are available to give an indication.

A UMLX diagram uses about twice the area of paper of NiceXSL5 written in a regimented style that might result from simple auto-generation from the diagram. It is more like four or five times the area when the NiceXSL is written in a more optimal manual style. With only a modest level of assistance from a GUI tool, a UMLX program probably requires more effort to enter but the UMLX diagrams involve considerable versions of schemas, for which a GUI tool can provide considerable assistance. The amount of information that needs entry on a UMLX diagram is therefore probably less than that of the equivalent XSLT, and of course much of the UMLX can be checked at compile-time or even edit-time, whereas XSLT leaves many loopholes for making debugging difficult.

In the absence of extensive ergonomic analysis, it is likely that a UMLX program will require more entry effort than XSLT, until a well-tailored GUI is available. But, the UMLX is a checkable specification that can exploit graphical layout. UMLX is more readable, genuinely re-usable and amenable to optimization. Since getting the UMLX bootstrap compiler to a level where transforms are validated as consistent schema instantiations, the author has found increasing benefit from resolving the intended transformations at the UMLX level, which then makes for a relatively simple

transliteration into XSLT with much reduced trouble with XPaths. Particular benefit comes when the schema is changed, since all affected transforms are diagnosed automatically. Willink further surmises that UMLX is a natural extension of the unified modeling language. It has good composition properties such as arbitrary OCL at the innermost level, declarative transform concurrency and declarative transform sequencing.

Willink (2003) asserts that the increased use of modeling techniques has led to the use of models as a programming language in model driven architecture. This has led to the exploitation of XMI for model interchange and XSLT for model transformation. It is a graphical transformation tool with minor extensions to UML but uses high level languages for transformation. It has the same concepts as UML. UMLX uses standard UML class diagrams to define information schemas, their instances and extends the class diagram to define inter-schema transformations. Hierarchical transforms have invocation contexts established by binding left hand side contexts to input ports and right hand side contexts to output ports. The multiplicities of the transformation determine the matching complexity.

A compiler is used to transform UMLX diagrams while specifying an application into a suitable application in a more visual descriptive manner. The incoming compiled transformation consists of an annotated copy of the schema for the user models together with a model each for the building and scanning phases. These three models are used by the main transformations: Build, Scan and Annotate Usage. Every transform and incoming schema is annotated with properties that help in the production of build and scan models by the Create Build and Generate transforms. An annotation identifies the schema composition arc instantiated by a composition arc in a transform. This is important because though the configuration of GME to support editing of UMLX

diagrams ensures that class instances are associated with defining classes, it does not validate the correspondence of relationships. This transformation forms part of an editor validation to improve the context of diagnostics and their immediacy. UMLX uses schemas for the user models and transforms and specifies what needs to be done. UMLX defines transformations as components which can be composed in a hierarchical manner but unlike normal component hierarchies which are statically defined and used repeatedly, UMLX component hierarchy are dynamically determined by matcing the input data and then used exactly as the single execution of the transformation produces as many times as possible.

An Analysis of VMT

Sendall et al. (n.d.) describes the VMT approach proposes a principally visual language for describing transformations between models specified with UML. In the language, a transformation is defined in terms of a set of transformation rules. Each transformation rule defines the way that one or more target UML diagram elements are created, changed, and/or deleted as a function of zero or more source diagram elements. This process can be seen as a mapping from source to target diagram elements, where target elements may be created in the process. A transformation rule is elaborated by a rule specification. A rule specification consists of two parts: a matching schema and a result schema. The matching and result schemas are inspired from the work on graph transformation approaches. The matching schema of a rule specification defines the condition under which the rule has permission to fire, the input arguments for the transformation, and those input arguments that will be deleted with the execution of the rule.

A result schema defines the target diagram(s) as a function of the elements bound by the matching schema. A result schema is also represented as a graph. Intuitively, a node in the result graph represents an element of a target diagram. In particular, a node can represent a newly created element, a modified element from the source model, or an unmodified element from the source model. Like for the matching schema, it is also possible to have a node that represents a set of target model elements.

Lapeyre and Usdin (2006) argue that XSLT reads XML documents and writes HTML for browsers, a flat ASCII file such as commas and plain text, interchange files such as EDI, RDF and RTF, typesetting driver files such as Frame Maker, QuarkXPress and InDesign or a different XML tag set. The XSLT engine reads XML documents including tags and text and uses an XSLT style sheet to transform by running the XSLT processing software which is an XSLT engine to produce the output documents.

Olteanu (n.d.) shows that XSLT was primarily designed to transform the XML document structure. This involves selecting, projecting-joining, aggregating, grouping and sorting data and then formatting the data into the desired output structure. Clark (1999) asserts that XSLT transformations are attained by associating patterns with templates. It uses the expression language X Path to select elements for processing to generate test. Tilton (2001) argues that the XSLT processor instantiates templates to generate the result document. Villard and Layaida (2002) argue that incremental transformation processors like incXSLT provide a better alternative to aid design of the transformation sheets and content. According to Bittner (2004), the data mapping tool prototype Clio automatically generates XSLT style sheets. He argues that it uses a common means for XML data transformations which is done using the XSLT processor.

An Analysis of MOLA

Sostaks, Kalnina, Kalnins, Celms, and Iraids (2011) assert that MOLA is a graphical transformation tool that combines declarative pattern specification and imperative control structures to determine the transformation execution order. The major element of MOLA is a rule. It contains the declarative pattern that specifies instances from which classes are selected and linked. Kalnins, Celms, Sostaks (2004) argue that MOLA is a graphical procedural transformation tool with distinguishable advanced graphical pattern definitions and control structures. Just like many other model transformation languages, it is based on the source and target metamodels. The MOLA tool consists of the transformation execution environment and the transformation definition environment. They use a common runtime repository which is a relational database.

According to Kalnina, Kalnins, Sostaks, Iraids, and Celms (2011), MOLA is based on the concepts of pattern matching and rules defining the way the elements of the matched pattern should be transformed. MOLA programs transforms instances of source metamodels prescribed in the MOLA meta-modeling language into instances of the target meta-model. MOLA has an eclipse-based graphical development support. Kalnins, Celms, and Sostaks (2005) show that MOLA has two parts that is the definition environment and the execution environment. The MOLA definition environment is based on the generic modeling framework and has graphical editors for MOLA diagrams, metamodel graphical editors and the metamodel compiler.

An Analysis of Rational Rhapsody

Four languages are supported in Rational Rhapsody they are Java, Ada, C, and C++. Rational Rhapsody uses deployment, component, collaboration, structure, flow,

activity, state chart, sequence, use case and object model diagrams in its development environment. Rational Rhapsody can export or import generated codes, XMI files, requirements, diagrams, models and projects from other modeling tools. It is a stand-alone interface that is used to create model items while showing the relationship between those items. Figure 8 shows relationships between items in Rational Rhapsody.

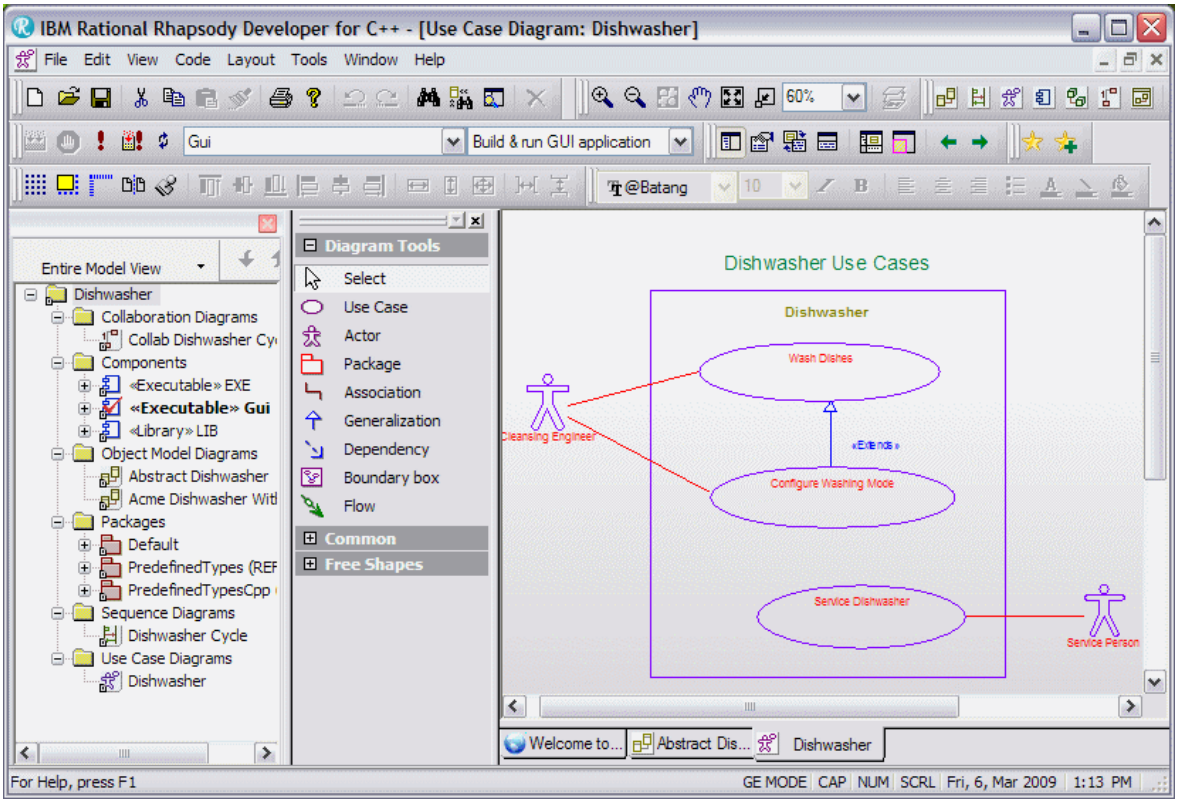


Figure 8: Relationships between items in Rational Rhapsody. From: IBM Corporation. (2009). Rational Rhapsody Getting Started Guide. Retrieved November 1, 2012, from http://publib.boulder.ibm.com /infocenter /rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/ pdf75/Getting_Started_Guide.pdf

When the models are being tested with the code menu, run icons, make and generate options, Rational Rhapsody displays log outputs and messages in the output window. It groups different information types including search results as in Figure 9. Rational Rhapsody displays the generated material in a separate window if the model requires an interface or messages to be displayed.

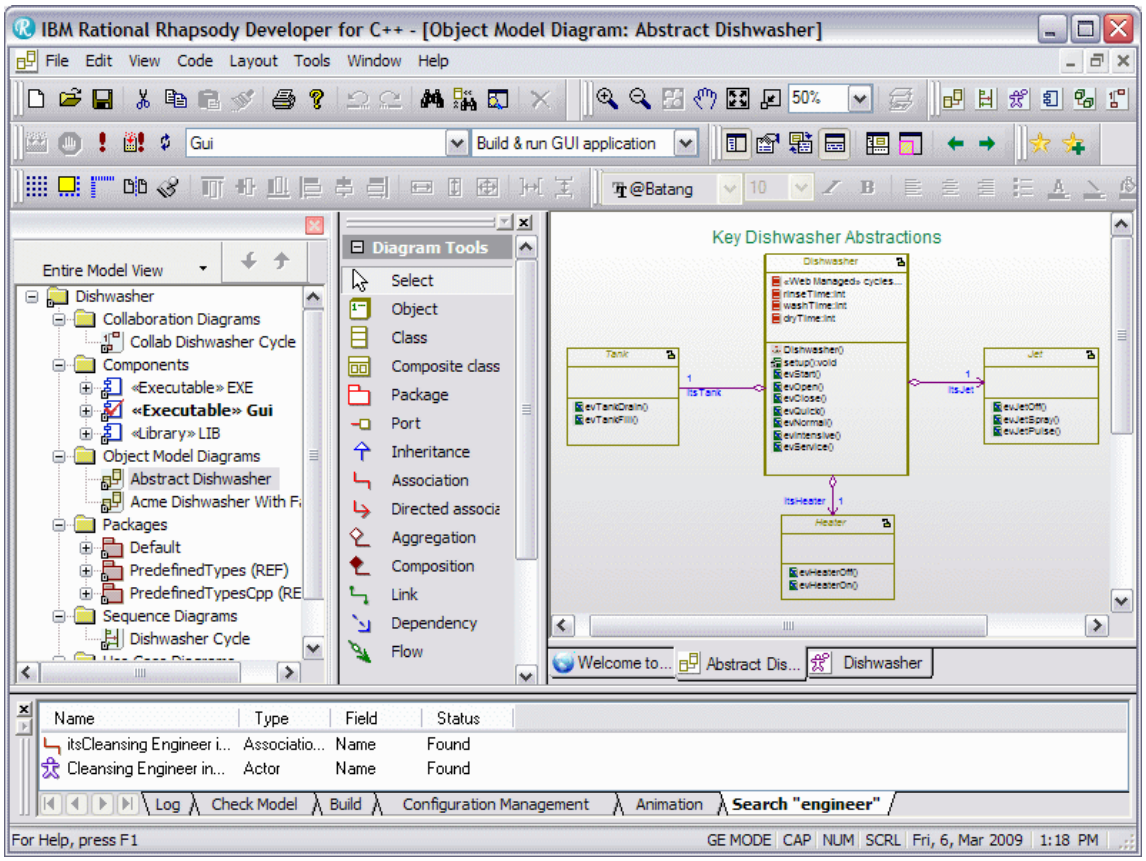


Figure 9: Grouping of information in Rational Rhapsody. From: IBM Corporation. (2009). Rational Rhapsody Getting Started Guide. Retrieved November 1, 2012, from http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/Getting_Started_Guide.pdf

Evaluation Criteria of the Model Transformation Tools

The three tools compared and analyzed were the MOLA, BOTL and IBM's Rational Rhapsody model transformation tools. This is because they have more visible characteristics and distinct attributes. They have distinct and more observable modes of operation and cover all model transformation aspects.

Transformations were observed UML-UML in the noted tools. The qualities of transformation tools analyzed include popularity in usage, graphical notations, lexical notation, declarative nature, bidirectional, inheritance in terms of updating and maintain transformations and XML support. The tool requirements analyzed as derived from the discussion by Grønmo and Oldevik (n.d.). These include: Text-to-UML (reverse engineering), UML-to-UML, UML-to-text which is needed to implement running system

as well as for documentation purposes, UML tool independence which is desirable so that the solution is not tied to a single UML tool which will be observed and compared in the noted transformation tools.

Other tool requirements such as absence of proprietary intermediate structures which increase the complexity for the transformation architect, traceability, metamodel-based that is the source and target metamodels which are explicitly defined and exploited in order to drive the transformation specification.

The tools that were compared are metamodel-based. This means that the source and target metamodels are defined and exploited to drive the transformation specification. The meta-model based aspect ensures that transformation specifications are correct. The dependability, usability and robustness of the selected tools are distinct in the selected tools. They also have elaborate functionality and architecture. The tools I use have traceability quality. Traceability is the ability to provide explicit traces of every target element back to the corresponding source elements. This enables us to understand the transformation and modifying the transformation and the source model to get better results. The selected tools have no proprietary intermediate structures. This is good because intermediate structures increase the transformation complexity. The selected tools have UML tool independence which is suitable because solutions are not tied to the single UML tool which enhances the comparison parameters. They also have the ability to implement UML-to-text transformations which enhances code generation to implement running systems and enhance documentation. They have the ability to perform UML to UML transformations which facilitates transformations between platform specific and platform independent model.

BOTL and Rational Rhapsody are able to perform Text to UML transformations which imply the ability to specify reverse engineering transformations from text or code to UML models. This improves documentation in model-driven settings. Properties like XML support, Bi-directionality, declarative nature, lexical notation, graphical notation, inheritance and their level of usage are thought to be able to observe, analyze and compare. As derived from the research paper by Wang (2005), other parameters that will be used in comparing the tools include the tooling aspect. This is whether the tools have a strong tooling support. The robust of the transformation execution in the sense that the execution of the transformations can continue even with execution errors will be evaluated. The ability of the tools to support reuse and composition will be analyzed as well as the ease of understanding of the tool's documentation and the graphical user interface. The availability of an abstract syntax for the transformation language for imperative, declarative as well as composition parts will be evaluated.

Chapter 4: Outline of Completed Project

Chapter 4 presents the findings of the chosen tools, their operational procedures as well as various transformation outcomes. The three tools considered are MOLA, BOTL, and Rational Rhapsody Developer Tool. Two of the chosen tools, MOLA and BOTL were chosen based on their ease of use as determined by setup time and learning curve. Rational Rhapsody Developer was chosen as the third tool because it is a commercial product and may be more in line with a tool of interest to a larger community. As Rational Rhapsody Developer tool is the most functional of the three, a detail walk through of the transformation outcomes of the tool is presented. Table 3 presents a concise representation of the outcomes, based on the attributes the tools were measured against.

Findings and Evaluation Criteria of Model Transformation Tools

The methodology in Chapter 3 provides an analysis of various transformation tools discussed in Chapter 2, based on their literature reviews. The various transformation tools described identify them all as tools that are UML based. Within a software development environment standardization and the ability to refine high level models into artifacts are critical for the acceptability of a transformation tool. Transformation tools should possess theoretical properties such as completeness, correctness, and termination. A transformation tool should also be scalable to cope with complex and large projects. The basic operational procedures of three of the tools as well as their transformation outcomes are presented as follows:

MOLA Tool

The MOLA tools consist of two parts:

- Transformation Development Environment (TDE)

- Transformation Execution Environment (TEE)

However, MOLA widely use Java compiler version, API of Eclipse EMF used for model transformation.

MOLA is a model transformation language using a simple procedural control structures. Typically, MOLA uses a graphical loop concept to achieve its designing function. Using a set of graphical editors, MOLA could support graphical diagram for building a system. MOLA has been used for some large scale projects and one of the projects is the "6-th Framework IST project ReDSeeDS" using MDA guidelines.

"Non-trivial transformations are used to obtain the initial version of Architecture model (in a rich subset of UML) from the requirements model. Similarly, the Architecture model can be transformed to a Detailed Design model (analogous to OMG PSM). The first versions supporting MOLA transformations have already been built and tested in practice. Here, the JGraLab repository is used, since some other tools in ReDSeeDS also use this repository" (Kalnins, Sostaks, Kalnina, et al 2009).

BOTL Tool

BOTL is a mathematical transformation tool that can easily be extended to specify transformations on a single model. The tool offers a protocol for the description of tool chains and model integration. BOTL is based on formal comprehensive graphical notation using UML-like notations.

BOTL uses a formal foundation, compressible, precise, and graphical notation to achieve a transformation objective. Analysis of BOTL reveals it is not a commonly used language. BOTL uses a mathematical foundation and graphical description techniques to achieve a set of mapping rules transformation mechanisms to deliver a transformation (Braun, and Marschall, 2003). However, the shortcoming of BOTL is that it is not a

commonly used language and it does not have an inheritance feature to make a quicker transformation.

Rational Rhapsody Developer Tool

This section presents the work by IBM Corporation of the Rational Rhapsody Getting Started Guide operational procedure. Transformation outcome of Rational Rhapsody Developer tool is considered.

A system architect, software developer and system engineer use Rational Rhapsody as an embedded and real time system. Rational Rhapsody uses a visual design environment to create model systems using UML diagrams. Rational Rhapsody assists developers in constructing a system using any of the four supported languages, which include Java, Ada, C, and C++. The operational procedure is implemented using the following procedures: First, a developer or system engineer will need to download one of the following Rational Rhapsody product families.

- Rational Rhapsody Architect for a Systems Engineer
- Rational Rhapsody Designer for a Systems Engineer
- Rational Rhapsody Architect for a Software Developer
- Rational Rhapsody Developer.

When a developer decides to create a new project using Rational Rhapsody it is essential to follow the following steps:

- “Navigate to the Rational Rhapsody <version>\Samples directory in your installation to examine the official Rational Rhapsody sample models”.
- “Click Proceed in the Project Samples area of the Welcome screen”.
- “Click the Samples icon on the Welcome screen” (IBM 2009 P 2).

One of the most important operational functions of Rational Rhapsody is its ability to create Standard UML diagrams. Rational Rhapsody assists a system engineer, information architect and software engineer to create the following UML diagram in the development environment:

- Activity diagram
- Component diagram
- Collaboration diagram
- Deployment diagram
- Flow chart
- Object model diagram
- Sequence diagram
- Statechart
- Structure diagram
- Use case diagram

Rational Rhapsody also assists in designing additional specialized diagrams that include:

- Block definition diagram
- Requirements diagram
- Parametric diagram
- Internal block diagram

The package contains a sample of an activity diagram in the following Rational Rhapsody installation folder: “Samples\SystemSamples\ActivitiesDiagramSimulation.”

In the Rational Rhapsody package, it is possible to stimulate the activity diagram to test the new features. The activity diagram could be created automatically, stimulated and run as class activities.

Procedure

- Start Rational Rhapsody software
- Navigate to the Packages, Use cases, Blocks or Operations, to the activity diagram
- Right click the component of the package, or operation to add the activity diagram
- Select Add New > Diagrams > Activity Diagram

Alternatively, there is an Activity Diagram icon located at the top of the window. Figure 10 reveals the operations of the activity diagram discussed.

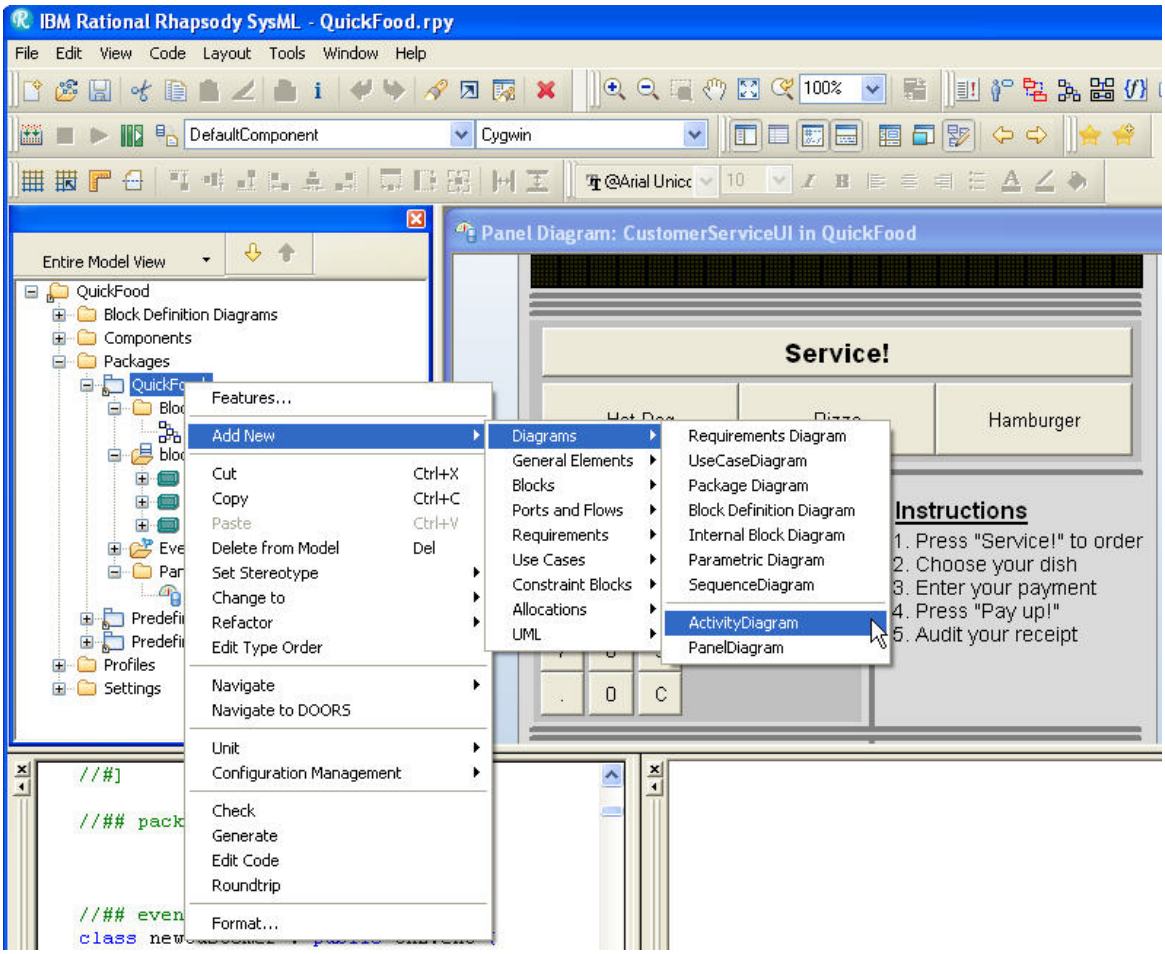


Figure 10: Activity Diagram. From: IBM (2009). Creating activity diagrams. IBM Corporation.USA. Retrieved February 20, 2013, from http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp?topic=%2Fcom.ibm.rhp.sysml.doc%2Ftopics%2Frhp_t_dev_creating_activity_dgrms_sysml.html

Results

Based on the above operation, Figure 11 shows the result that is automatically generated of adding a new activity diagram.

Apart from assisting the system designer, system engineer, or information architect to create a dynamic UML diagram, Rational Rhapsody also assists in code generation. This study analyzes the operational procedure in the code generation strategy using Rational Rhapsody.

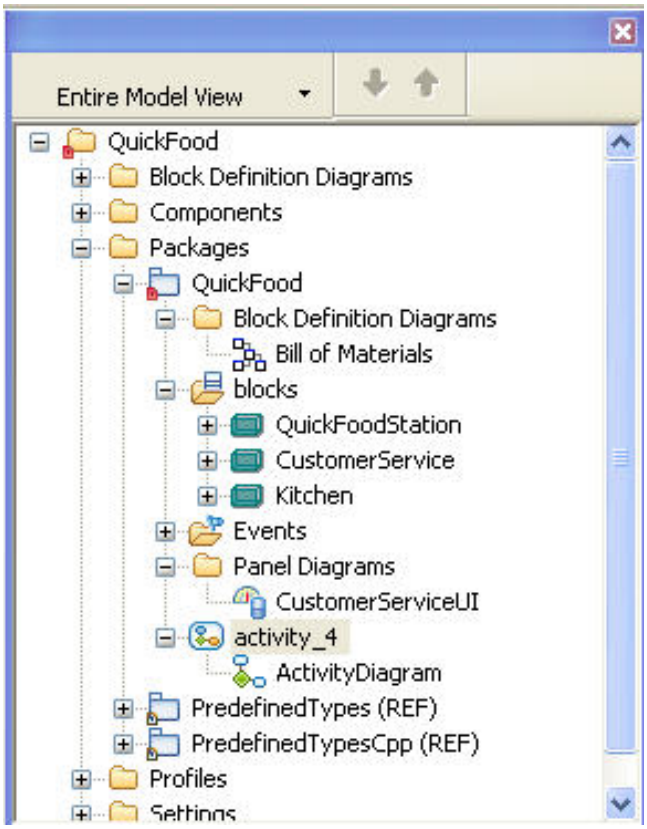


Figure 11: Result of Adding New Activity Diagram in Rational Rhapsody. From: IBM (2009). Java Tutorial for Rational Rhapsody. IBM Corporation.USA. Retrieved February 20, 2013, from http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/tutorialj.pdf.

Rational Rhapsody Java Code Generation

The study presents the strategy to generate code from the Dishwasher project. To create a Dishwasher project, it is essential to provide the analysis of the Dishwasher system.

Dishwasher System Analysis

The following questions are considered through the analysis of the Dishwasher Project:

- Who are going to use the system?
- How they are going to use it?
- What are the more important actions of the system?

- When are these actions going to occur?
- What are the relationships, differences or similarities between the actions?
- What are standard behaviors?
- What are the actions that can go wrong?

Based on the questions specified above the following specified answers are considered:

- The system actors or users would include “service” and “user” person.
- The system rinses, washes, and then dries dishes.
- The “user” will load the dishes into the dishwasher. The user will then start the dishwasher, and finally remove the dishes after they are washed.
- The system might fail to rinse, wash, or dry the dishes and will require additional service.

Thus, three types of actors are identified:

- System’s users,
- External components responsible to provide information to the system,
- External components that are receiving information from the system

The next step is to create a Use Case Diagram for the dishwasher system revealing the interaction of the system with the external actors considering interacting with the Dishwasher. The diagram below reveals the Use Case Diagram of the Dishwasher.

Creating a Statechart

Statecharts for Dishwasher project defines the life cycle behaviors of reactive classes and include various states that cause transition between one another. Figure 12, the diagram below, reveals the use case diagram for the Dishwasher project.

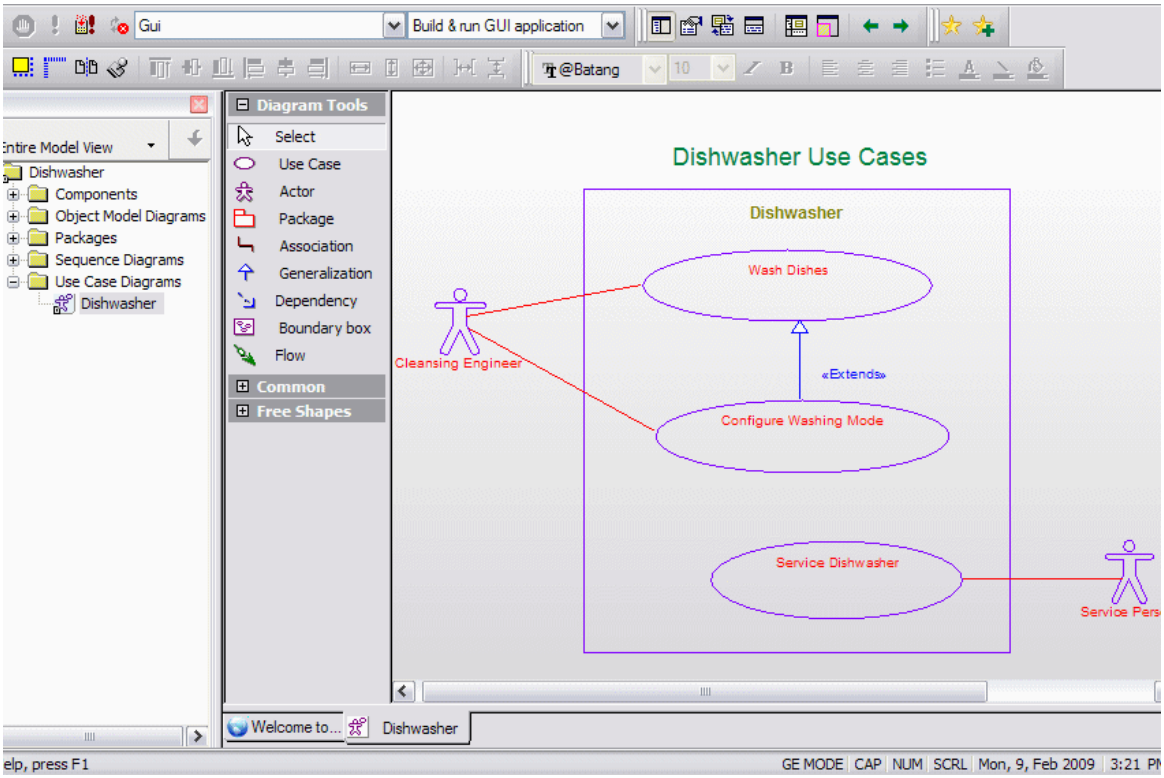


Figure 12: Use Case Diagram #1 for Dishwasher. From: IBM (2009). Java Tutorial for Rational Rhapsody. IBM Corporation.USA. Retrieved February 20, 2013, from http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/tutorialj.pdf

Object Model Diagram

Building the Object model diagram reveals the type of the objects in the system, and operations.

Creating Object Model Diagrams (OMDs)

The next step is to create the Object model diagrams that:

- specify the attributes, the type of objects in the system,
- operations belonging to those objects,
- the static relationship that could exist between types of classes and,
- the constraints that might apply.

From the OMD, Rational Rhapsody generates code that translates the OMD model into Java source code. Figure 13 reveals the Object model diagram for the Dishwater project.

Generating Code

The code generator of the Rational Rhapsody translates the element of model into OMD source code. The Java code for the Dishwasher project is presented in Figure 14.

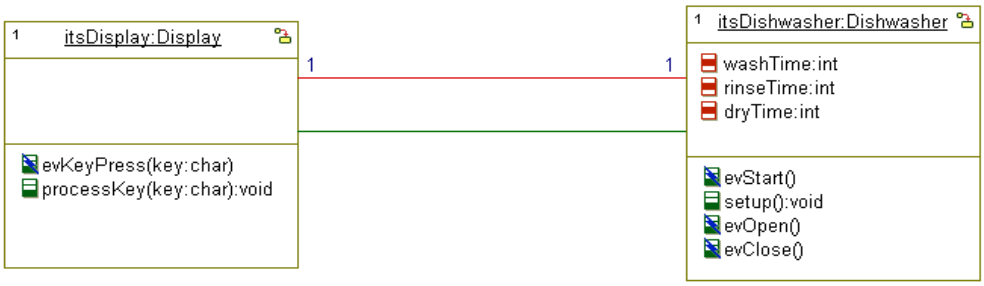


Figure 13: Object Model Diagram for the Dishwasher Project. From: IBM (2009). Java Tutorial for Rational Rhapsody. IBM Corporation. USA. Retrieved February 20, 2013, from http://publib.boulder.ibm.com/ infocenter/rsdp/v1r0m0/topic/ com.ibm.help.download.rhapsody.doc/pdf75/tutorialj.pdf

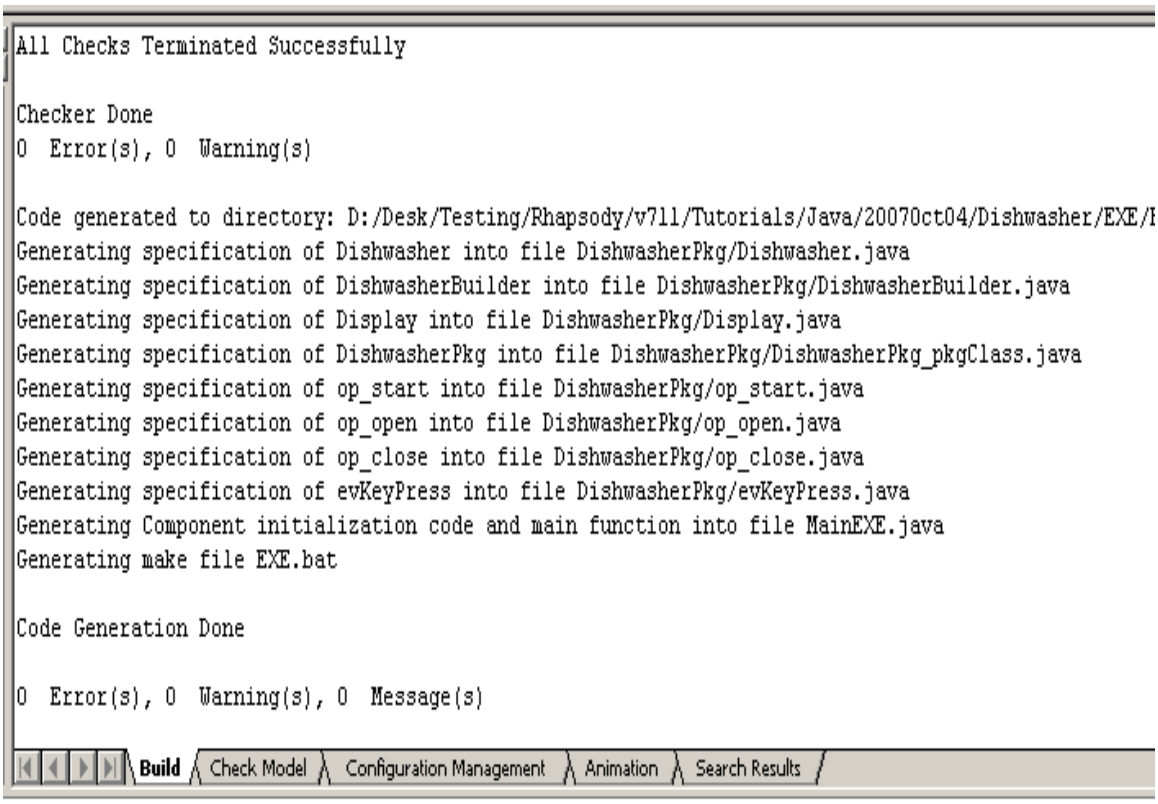


Figure 14: Java code for the Dishwasher Project. From: IBM (2009). Java Tutorial for Rational Rhapsody. IBM Corporation.USA. Retrieved February 20, 2013, from http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/tutorialj.pdf

Chapter 5: Discussion

This chapter presents a discussion of the benefits of UML, and it discusses the analysis of the tools in the comparative approach. The study also delivers an evaluation of the selected transformation tools.

Analysis of the transformation tools discussed in this project reveals that they base their design methodology on Unified Modeling Language, which is a standard graphical notation and language to specify, visualize, document and construct the artifacts of software systems.

> "The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing objects oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects" (Wang, 2005).

UML assists developers to communicate, explore and validate a software architectural design. More important, UML provides an abstract language to describe a graphical notation of software systems, (Wang, 2005). Based on the importance of UML, the study evaluates each of the transformation tools to enhance greater understanding of their strengths and their weaknesses. Grønmo and Oldevik (2005) uses different criteria in evaluating the language tools:

- "Is it possible to identify a set of fairly objective and easily checkable requirements that measures the quality of a model-driven transformation tool"?
- "Tool-dependant vs. language-dependant testing"
- "Ease-of-use? Robustness? Pricing?" etc.

- "Narrowed model scope in this study" (Grønmo and Oldevik 2005).

An evaluation of language tools by Grønmo is presented in Appendix C. Analysis of all the transformation tools presented in Appendix C and introduced in this study suggests that Rational Rhapsody has high quality features compared to other transformation tools. In the present business environment, companies are increasingly demanding an advanced software product that can deliver the latest technology. Today, the upsurge in a design's complexity, shrinking design cycles, and limited financial resources have all combined to create challenges in the design environment. UML provides a significant backbone to the development of language tools. The model transformation tools deliver benefits such as model-to-model, code generation, and reverse engineering. A transformation architect could use one of these tools to design a dynamic software system.

Despite the benefits derived from the operation of MOLA, the MOLA language is not a commonly used language and is more suited for academic purpose. Moreover, MOLA does not have an inheritance features that allows for quicker transformation. Additionally, MOLA does not possess other transformational requirements such as lexical notation, declarative features, bidirectional and XML support.

MOLA is a graphical transformation language similar to BOTL, UMLX and Rational Rhapsody. MOLA uses the traditional concept of transformation language to combine the imperative with the declarative style in its programming pattern. (Kalnina, Kalnins, Celms, et al, 2010). To its benefit, MOLA has important features such as supporting Text-to-UML, and UML-to-UML. However, MOLA is not as strong as other transformation tools such as Rational Rhapsody and XSLT, because MOLA does not support many important applications such as Lexical notation, UML-to-TEXT, XML

support, UML tool independence, Declarative, and Bi-directionality. Typically, MOLA is not a commonly used language such as Rational Rhapsody and XSLT because MOLA does not support complex transformation scenarios and does not support many dynamic applications that could make it universally acceptable by developers.

BOTL uses a formal foundation, compressible, precise, and graphical notation to achieve a transformation objective. Analysis of BOTL reveals it is not a commonly used language. BOTL uses a mathematical foundation and graphical description techniques to achieve a set of mapping rules transformation mechanisms to deliver a transformation (Braun, and Marschall, 2003). However, the shortcoming of BOTL is that it is not a commonly used language and, like MOLA, does not have an inheritance feature.

The BOTL Tool is ArgoUML4BOTL: BOTL is from 'Editor for BOTL Meta rules and models.' This tool also produces BOTLXML specifications.

The process of installation is to download the software from its home page, which is located at the following website, http://www4.in.tum.de/~marschal/botl/download.htm, and run the download jar file.

The main features are:

- BOTL is based upon a formal, precise, graphical and a comprehensible notation.
- BOTL comes with techniques to allow a developer to verify whether a BOTL specification could be used to produce models conforming to a given meta model.
- BOTL's specifications are bijective.
- The prototypical model transformation of BOTL tool is based on ArgoUML.

The category of BOTL is as follows:

- “BOTL is PIM to PSM Transformation Tool”
- “PIM to code Transformation Tool”
- “PSM to code Transformation Tool”
- Tunable Transformation Tool
- Transformation Definition Tool

The following criteria deliver the transformation tools features of BOTL:

Self-containing: Many documents are available in BOTL and BOTL introduces concept in the documents.

Scalability: It is not possible for user to freely decide the dimension of the transformation.

Simplicity: Transformations are designed automatically for users.

Bi-directional mappings: Theoretically possible because the works on bijective is currently still under construction.

Ease of Adoption: the mechanisms rules for BOTL is well defined, could be used as transformation tool set in the other projects.

Rich Conditions: BOTL does not support View/Query/Transformation

Abstract syntax language: BOTL does not support abstract syntax for the transformation language.

Adopt common terminology: BOTL document is understandable because it is written in English.

Reuse and Support composition: BOTL does not support repository.

BOTL supports complex transformation scenarios.

Provide complete examples: BOTL provides many examples of class diagrams.

Robust transformation executions; A robust transformation executions is not yet supported.

- Tooling aspect: BOTL does not contain a tooling aspect.
- Figure 15 provides an example of Java code put into XML interface.
- Input format: Botl file, XMI, Object Diagram(XMI), XML,
- Output format: Botl file, XML, java file.

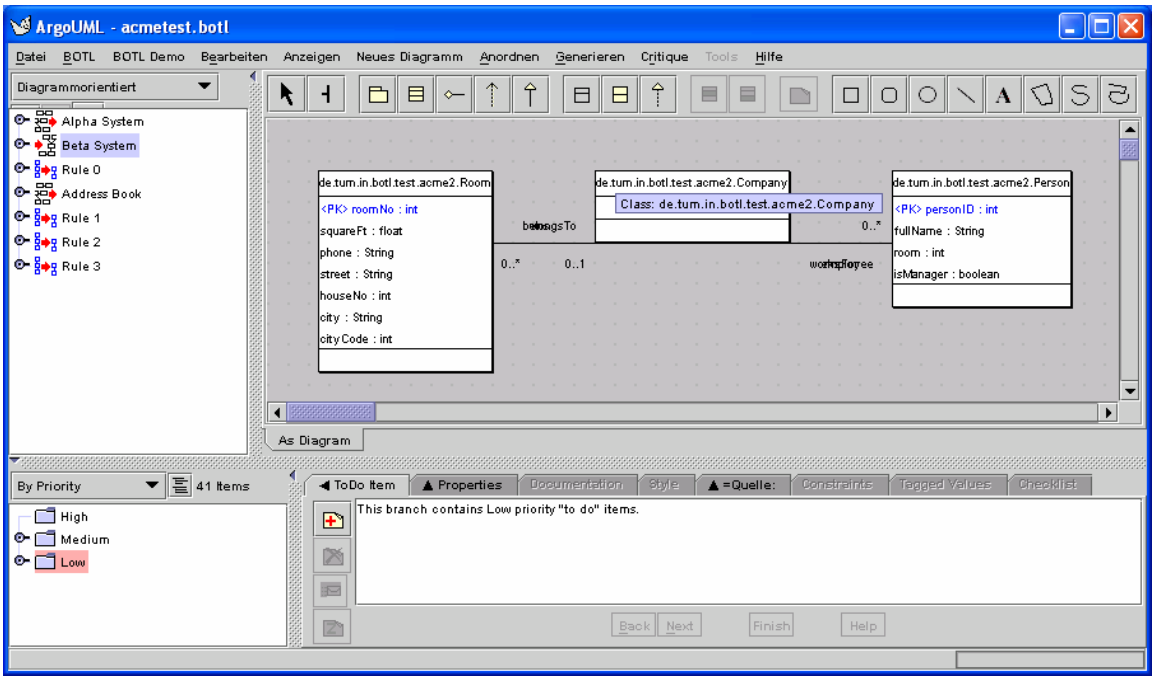


Figure 15: XML input into Java Code. Grønmo,R. andOldevik, K.(2005). An Empirical Study of the UML Model Transformation Tool (UMT). First International Conference on Interoperability of EnterpriseSoftware and Applications Retrieved February 20, 2013, from http://www.big.tuwien.ac.at/system/theses/61/papers.pdf?1298558494

Despite the benefits derived from the BOTL application, BOTL has some shortcomings. BOTL has not been able to fulfill the requirements defined within the MOF 2.0, QVT. Moreover, BOTL is still only concentrating on class diagrams in the Object oriented design. The input of BOTL is only in Java, and its XMI is still under construction.

IBM Rational Rhapsody delivers a MDD (Model Driven Development) that can assist a developer to automate the software development process as well as identify designing errors and defects in the development lifecycle. Rational Rhapsody assists a software developer or system engineer to deliver innovative and high quality software designs. The development process within Rational Rhapsody assists a developer to create and test code accurately and quickly. Based on the analysis of their operations, Rational Rhapsody Developer tool is able to satisfy excellent operational requirements and able to offer the transformational outcomes needed.

Rational Rhapsody is UML based and uses graphical models to generate a software application. Rational Rhapsody assists model checking, model verification, model execution and stimulation. A special benefit of Rational Rhapsody is that it assists in code improvement and it enhances the visual design environment for creating unified modeling language diagrams. Rational Rhapsody generates a complete application code from the state charts, activity diagrams, class diagrams and flow charts that supports UML 2.1.

The transformation outcome is the results of the model at the same and different levels of abstraction. The quality design and transformation outcome is also based on:

- Ease of use
- Ability to deliver program from model to code
- Ability to implement a program in a logical sequence of choices

The study uses a high quality transformation language to evaluate the transformation outcome of Rational Rhapsody. Table 2 summarizes the tool requirements Rational Rhapsody fulfills.

Table 2 - *Rational Rhapsody Transformation Tool*

Requirements	Rational Rhapsody	Discussions
Commonly used language	Yes	Rhapsody was released in 1996. Rhapsody is a well-received tool that is highly useful for developers.
Inheritance	Yes	Rhapsody has ability to make a quick new transformation as well as making changes relevant to many transformations.
Graphical notation	Yes	Rhapsody has ability to specify transformations Graphically. The model also uses UML-to-UML transformations to make graphical models. The higher-level view enhances the communication of the lexical counterpart.
Lexical notation	Yes	Rhapsody can specify the transformation lexically. It has ability to handle complex graphical notations having a scalability problem.
Declarative	Yes	Rhapsody is able to support statements that the transformation could do. The importance of declarative is that it does not operate on the computer system. This feature makes the declarative language to be side-effect free and reduces errors in the written code.
Bidirectional	Yes	Rhapsody supports bi-directionality. The advantage is that bi-directional language and specification are easier to maintain.
XML support	Yes	Rhapsody provides built-in support to produce or consume XML, which include ability to support general XML, attributes, elements, and namespaces.
Text-to-XML	Yes	With reference to Text-to-UML, Rhapsody has ability to specify reverse-engineering, transformations from a code/text to UML models. A graphical UML model can improve documentation..
UML-to-UML	Yes	Rhapsody has the ability to transfer transformations from a UML model into another UML model. The benefit derived from this feature is to transform platform independent models into platform-specific models.
UML-to-TEXT	Yes	Rhapsody can specify transformations from a UML model to text format. The benefit derived from this feature is to achieve code generation as well improved documentation.
UML tool Independence	Yes	Rhapsody uses UML tool independence to support transformation of models into other UML tools.
No Proprietary Intermediate Structures	Yes	Rhapsody has been able to use this feature to reduce complexity for a transformation architect.
Traceability	Yes	Rhapsody has the ability to provide explicit traces on every target element. This feature assists a developer in understanding the source model and modification of the transformation to get better results from the software system.
MOF-based /Metamodel	Yes	Rhapsody uses MOF-based /Metamodel to get a better results on to the transformation specification. Using the Metamodel-based language, a developer could be able achieve a correct transformation as well as checking whether the metamodel follow the explicit transformation rule.

Table 2: Discussion of the Rational Rhapsody requirements being fulfilled.

Rational Rhapsody assists developers deliver a robust set of modeling to produce high quality code to meet a stakeholder's requirements. Rational Rhapsody assists in achieving:

- Stimulation, model verification, and execution capabilities,
- Static model checking that assists a developer to verify the completeness and consistence of models,
- the support for object based, functional, or object-oriented paradigms.
- the integrated requirements modeling that assist a developer to design software that meets stakeholder's requirements,
- the ability to integrate an external code within the modeling environment,
- Domain-specific language to enhance support for AUTOSAR, graphical C, Modeling, Embedded (MARTE) systems, Analysis of Real-Time.
- Automatic diagram creation that assists in visualizing the existing code. (IBM, 2009).

Rational Rhapsody has embedded features that deliver the following quality software environment:

- Harmonizing software development and systems engineering with SysML and UML modeling,
- Ability to generate Java, C, C++, and Ada applications as well as behavioral and architectural views,
- Providing a model-level debugging via model execution that generates real-time sequence diagrams and highlights state charts during execution,

- Includes a requirements traceability capabilities and a robust set of modeling to enhance effective software delivery system,
- Integrating real time operating system as well as integration of industry-leading IDE (IBM, 2009).

Software Engineering Project Test Case in Rational Rhapsody

The following is a short test case designed to generate skeletal code from UML diagrams in Rational Rhapsody Developer.

The BirdCage Corporation needs software to aid the management of its business. The corporation builds its business around domestic birds. BirdCage arranges its business around different services with each service being managed by a different company making BirdCage a group of companies. Companies are divided into departments. Services include veterinary services, safekeeping of birds, bird calisthenics and a bird competitions service. In order to streamline its business, BirdCage classifies the veterinary and safekeeping services as medical services. The other two services are regarded as entertainment services. In the future the company hopes to add a number of "Nutritional" services to address the production and sales of food for birds. The corporation classifies birds as either exotic or non-exotic. BirdCage does not provide entertainment services for exotic birds. Owls and crows are non-exotic birds, while macaws, cockatoos and green quakers are exotic. BirdCage seeks to maintain a minimum stock of at least 10 of each type of exotic bird.

Rational Rhapsody successfully generated UML text files and skeletal code for project BirdCage from UML Diagrams. Full details and results can be viewed in Appendix D.

The Comparative analysis, as detailed in Table 3 and summarized in Table 2 reveals that Rational Rhapsody can deliver operational and transformational outcome with satisfactory effectiveness for the developer. Oldevik and Grønmo (2003) argue that the quality of the transformation language is critical in producing effective and efficient software development, and it can be observed in all the tools high quality language development.

The model requirements identified in Table 2 is desirable for a complete transformation tool; a tool that will deliver scalable programs and robust program code in the ultimate analysis. A transformation or information architect requires many technologies to succeed in model driven development. For one, a developer requires good models to succeed in the design environment and beyond throughout the life cycle to ultimately deliver the code. Some transformation tools deliver ready-made code generation templates that are difficult to customize. Rational Rhapsody offers an effective transformation that can assist software developers to customize code based on a developer's specifications.

Table 3 compares MOLA, BOTL and Rational Rhapsody against a set of attributes of a well-rounded transformation tool. For MOLA and BOTL my analysis of these tools confirm they are still in the developmental stages and do not completely satisfy identified ideal requirements of a robust transformation tool. My analysis of Rational Rhapsody Developer finds it fully developed and can assist a system architect in building scalable software intensive, detail rich software programs through the development life cycle.

Table 3 - Comparison of MOLA, BOTL and Rational Rhapsody Developer Transformation Tools

Requirements	MOLA	BOTL	Rational Rhapsody Developer
Commonly used language	No	No	Yes
Inheritance	No	No	Yes
Graphical notation	Yes	Yes	Yes
Lexical notation	No	No	Yes
Declarative	No	Yes	Yes
Bidirectional	No	Yes	Yes
XML support	No	No	Yes
Text-to-UML	Yes	No	Yes
UML-to-UML	Yes	Yes	Yes
UML-to-TEXT	No	No	Yes
UML tool independence	No	Yes	Yes
No proprietary intermediate structures	Yes	Yes	Yes
Traceability	No	No	Yes
Metamodel/MOF-based	Yes	Yes	Yes

Table 3: An analysis of MOLA, BOTL & Rational Rhapsody Developer. Table based on: Grønmo, R. (2001). An empirical study of the UML model transformation tool (UMT). SINTEF: Norway. Retrieved October 31, 2012, from http://heim.ifi.uio.no/~roygr/INTEROP-ESA-2005.pdf

Chapter 6: Conclusion and Future work

Conclusion

The constantly evolving technologies of the modern era are dependent on complex systems to run them. These systems have proven to be progressively costly and unwieldy to design, develop, manage and modify. This is of vital importance to business leaders whose experience with code-centric programs of the past has resulted in project obstacles that have exorbitantly increased their systems development and maintenance costs. To better hold down development costs while managing the increasing complexities of modern enterprise systems, Model Design Engineering envisions gains in productivity and development outcomes by veering away from a code-centric approach during the development life cycle, to a more simplified and intuitive model driven approach whose aim is to evolve software as a chain of model transformations. The ultimate need for code is met in the model driven development approach as an ongoing effort to bridge the gap between problem and software implementation domains. The significance of this research is its study of current technology within the context of an ever changing technological landscape and to what extent modeling tools are able to address present software engineering concerns in its need for improvement in processes both in modeling transformation, and the tools themselves used to support this process.

The project collects primary and secondary data to achieve the following research objectives:

i. To identify various UML based model transformation tools that have clear attributes.

ii. To investigate and analyze several UML based model transformation tools

iii. To select 3 UML based model transformation tools and elaborate on their functionalities

iv. To create a table comparison of all tools that shows their functionalities and differences.

To achieve the research objectives, the project is organized in six chapters covering the following: Chapter one introduces the background of UML based model transformation, the project research objectives and some foresight on the model transformation tools that were discussed in this study. Chapter two presents the literature review that explores the past research on the various transformation tools discussed in this study. The review of literature has informed the research to identify the strengths and the weaknesses of all the transformation tools discussed in the study. Chapters 3 provide the methodology of the project, and a preliminary analysis of all the transformation tools discussed in the project. The project also identifies some key attributes that assist in comparing the the key tools to be selected for further discussion. Chapter 4 presents the results of the chosen tools as well as their operational procedures and various transformation outcomes. Chapter 5 presents the analysis and discussion of the selected transformational tools and their operations. The chapter 6 presents a summary of project work, the conclusion and future work to be considered.

Future Work

The basic premise of MDA is to create transformations originating from a platform independent model (PIM), described in UML at the highest abstraction level as to functionality and structure and system; and through a chain of transformations converted to a platform specific model (PSM) and then to code using a standardization such as QVT. (Kuznetsov, 2006)

By describing models through a set of meta-models, transformation amongst models is facilitated, resulting in code generation (Goede/Irizarry, 2008). Meta-models come into play as the specification of a well-defined language used to describe models.

Research has proven that although early tools exist to help this process along, many tools fall short of the ideal with language issues, debugging and integration considerations. The ideal tool was identified as one fully integrating all aspects of development in an MDA modeling and development environment.

With caveats it can be concluded that model transformation tools have become an effective means of a software development approach. However, the caveats are: to enhance effective and efficient software development, a chosen software development tool needs to be able to automate the creation and adapt to the evolution of project development. A tool must appreciate and thereby enhance model relationships and dependencies during maintenance throughout the software lifecycle. The project provides an analysis of various transformation tools that could assist software developers and information architects to develop dynamic software products.

The evaluation discussion in Chapter 5 serves to enhance a greater understanding of the strategy to improve upon transformation tools for business applications. This study analyzes eight transformation language tools, where three were selected for more detail study, each of the three representing a different approach to achieving model transformation. The ideal tool is a transformation tool that contains the applications requirements that could be used as a designing tool for software developers, information architects, system engineers and transformation architects. A number to transformation tools have several shortcomings that may not make them suitable to develop a complex software project. For example, VMT, YATL, and XSLT transformation tools do not have

graphical notation requirements that might assist a developer in generating code from graphical notations. Moreover, MOLA, YALT, XLST, and UMLX do not have inheritance requirements that could assist developers to reuse code of an existing object. MOLA, BOTL, VMT, and YALT transformation model do not have XML support that could assist developers in implementing data mapping in a transformation.

Rational Rhapsody transformation tool has all the transformation requirements necessary for achieving a robust transformation product. IBM Rational Rhapsody provides a model-derived driven environment that can assist developers to automate the software development process. Rational Rhapsody also assists developers to optimize team collaboration by facilitating the identification of defects and design errors in the development lifecycle.

Future works are required to improve on other transformation tools discussed in this study. Rational Rhapsody meets the requirements of a model transformation tool and can be used to develop complex large-scale projects. There is a need to integrate graphical notation in the transformation languages such as YATL, XSLT and ATL to enhance their efficiencies and thereby make them more commonly used languages among developers. There is also a need to improve on all the transformation languages and ensure that all the requirements below are integrated into them to again make these languages more commonly used among developers:

- Inheritance
- Graphical notation
- Lexical notation
- Declarative
- Bidirectional

- XML support
- Text-to-UML
- UML-to-UML
- UML-to-TEXT
- UML tool independence
- No proprietary intermediate structures
- Traceability
- Metamodel/MOF-based

Is MDE, today, through standardizations like QVT implementing model transformation, the silver bullet to software engineering development obstacles and grief? Some, like Rahmouni (2011), a rather modern study, believe models, "in most cases…are still confined to a simple documentation role instead of being actively integrated into the engineering process." I can hardly conclude from my research that the paradigm shift for software development, from code centric to model centric, has occurred, however, technological demands for complex and intense systems will continue, and with them voices continue to demand a better more intuitive process to construct software. Continued work on tool refinement and the adoption of languages by developers will increase and quicken the paradigm shift. The approaches are sophisticated, the tools are sophisticated. There is much to learn and continued education will be the norm under constantly changing technologies. I am convinced that larger projects that are typical today can benefit from an intuitive development process, better organization, better separation of concerns, and improved traceability. The end result of adopting model transformation will be development enhancements such as improved management of

development artifacts, reliable and timely modifications and streamlined maintenance practices; these will be the software engineering rewards to look forward to.

References

Abmann, U., Zschaler, S., and Wagner, G. (2005). *Ontologies, Metamodels, and the Model-Driven Paradigm*, Dresden: Institute for Software and Multimedia Technology.

Amelunxen, C. and Schurr, A. (2008). *Formalizing model transformation rules for UML / MOF 2*, Merckstrasse: Darmstadt University of Technology.

Bell, D. (2003). *UML basics: An introduction to the Unified Modeling Language*. Retrieved October 26, 2012, from http://www.nyu.edu/classes/jcf/g22.2440-001_sp06/handouts/UMLBasics.pdf

Bittner, T. (2004). *Performance Evaluation for XSLT Processing*. University of Rostock.

Braun, P. and Marschall, F. (2003). BOTL The Bidirectional Object Oriented Transformation. Language. Institut für Informatik Technische Universität München. Retrieved October 29, 2012, from http://www4.in.tum.de/publ/papers/TUM-I0307.pdf.

Broy, M. (2005). *Software and Systems Engineering*, BOTLzmannstr. 3D-85748 .

Clark, J. (1999). XSL Transformations (XSLT) 1.0, WWW Consortium (W3C).

Cremers, A.B., Alda, S., Rho, T. and Speicher, D. (2009). Chapter 13, Mapping Models to Code Object-Oriented Software. Construction. University of Bonn. German. http://www.iai.uni-bonn.de/III/lehre/vorlesungen/SWT/OOSC06/slides/13%20-%20Mapping%20models%20to%20Code.pdf.

Czarnecki, K. and Helsen, S. (2003). *Classification of Model Transformation Approaches*. Ontario: University of Waterloo.

De Goede, Irizarry. (2008). *Understanding Tool Requirement for Model Driven Architecture.* Opportunity Blueprint.

Eclipse Foundation. (2012). *UMLX Language*. Retrieved November 2, 2012, from http://www.eclipse.org/gmt/umlx/.

Einarsson, H.P. (2011). Refactoring UML Diagram and Models with Model-to-Model Transformation. Master of Science in Software Engineering, University of Iceland.

Exforsys Inc. (2007). *UML basics*. Retrieved October 27, 2012, from http://www.exforsys.com/tutorials/uml/uml-basics.html

Freire, A.P. and Fortes, R.P. (2005). Automatic Accessibility Evaluation of Dynamic Web Pages Generated Through XSLT. ACM.

Giese, H. and Wagner, R. (2009). *From model transformation to incremental bidirectional model synchronization*. Springer-Verlag.

Grangel, R., Bigand, M., and Bourey, J.P. (2010). Transformation of decisional models into UML: application to GRAI grids. *International Journal of Computer Integrated Manufacturing* 23 (7), 655–672.

Grønmo, R., Belaunde, M., Aagedal, J.O., Engel, K.D., Faugere, M., andSolheim, I. (2005). *Evaluation of the Proposed QVT Merge Language for Model Transformations*. Miami: INSTICC Press.

Grønmo, R., Pederson, B.M., and Olsen, G.K. (2009). *Comparison of Three Model Transformation Languages*. Oslo: University of Oslo.

Grønmo, R. and Oldevik, K. (2005). An Empirical Study of the UML Model Transformation Tool (UMT). First International Conference on Interoperability of Enterprise Software and Applications.

IBM Corporation. (2009). *IBM Rational Rhapsody Developer*. Retrieved October 30, 2012, ftp://public.dhe.ibm.com/common/ssi/ecm/en/rad14043usen/ RAD14043USEN.PDF.

IBM Corporation. (2009). Rational Rhapsody Getting Started Guide. Retrieved November 1, 2012, from .http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/ topic/com.ibm.help.download.rhapsody.doc/pdf75/Getting_Started_Guide.pdf.

IBM Corporation. (2009). Creating activity diagrams. IBM Corporation. USA.

IBM Corporation. (2009). Java Tutorial for Rational Rhapsody. IBM Corporation. USA.

Jouault, F., Allilaire, F., Bezivin, J., and Kurtev, I. (2008). *ATL: A model transformation tool* Nantes: University of Twente. Retrieved October 25, 2012, from http://www.ie.inf.uc3m.es/grupo/docencia/reglada/ASDM/Jouault08.pdf.

Kalnina, E., Kalnins, A., Sostaks, A., Iraids, J., and Celms, E. (2011). *Hello World with Mola*. Riga: University of Latvia. Retrieved October 29, 2012, from http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/solutions/online/ttc2011_submis sion_28.pdf.

Kalnins, A. and Vitolins, V. (n.d.). *Use of UML and Model Transformations for Workflow Process Definitions*. Retrieved October 31, 2012, from http://arxiv.org/ftp/cs/ papers/ 0607/0607044.pdf.

Kalnins, A., Barzdins, J., andCelms, E. (n.d.).*Model Transformation Language MOLA*. Retrieved November 2, 2012, from http://melnais.mii.lu.lv/audris/MOLA_MDAFA.pdf.

Kalnins, A., Celms, E., andSostaks, A. (2004).*Model Transformation Approach Based on MOLA*.University of Latvia.

Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sosstaks, A., andBarzdins, J. (2004).*Building Tools by Model Transformations in Eclipse*. Retrieved November 1, 2012, from http://www.dsmforum.org/events/dsm07/papers/kalnis.pdf.

Kalnins, E., Celms, E, Sostaks, A. (2005).*Tool support for MOLA*.Riga: University of Latvia. Retrieved October 31, 2012, from http://melnais.mii.lu.lv/audris/ MOLAtoolGRAMOTFin.pdf.

Koch, N. (2007), *Classification of model transformation techniques used in UML-based Web engineering*. The Institution of Engineering and Technology. Retrieved October 26, 2012, from Ebscohostdatabase (doi:10.1049/iet-sen:20060063).

Kuznetsov.(2006).*UML Model Transformation and Its Application to MDA Technology*.Moscow State University.

Lapeyre, B. and Tommie, U. (2006). *Introduction to XSLT Concepts*. Mulberry Technologies Inc. Retrieved November 1, 2012, from http://www.mulberrytech.com/papers/Intro2XSLT/Intro2XSLT.pdf.

Li, D., Li, X., Stolz, V. (2011). QVT-based model transformation using XSLT. *ACM SIGSOFT Software Engineering Notes*, 36(1). Retrieved November 1, 2012, from http://rcos.iist.unu.edu/index.php/qqvt-based-model-transformation-using-xsltq-acm-sen.

Marschall, F. and Braun, P. (2004). *Model Transformations for the MDA: BOTL*. New York: Columbia University Press.

Marschall, F. and Braun, P. (2003). *Model Transformations for the MDA with BOTL*. BOTLzmannstr: Institute for Information. Retrieved October 27, 2012, from http://www4.informatik.tu-muenchen.de/publ/papers/marschall_braun-mdafa03.pdf.

Marschall, F. and Braun, P. (2003). *Transforming Object Oriented Models with BOTL,* 72(3). Retrieved November 6, 2012, from http://www.sciencedirect.com/science/ article/pii/S1571066104806157.

Mens, T., Czarnecki, K.,and Gorp, P.V. (n.d.). *A Taxonomy of Model Transformations*. Retrieved October 28, 2012, from http://drops.dagstuhl.de/volltexte/2005/ 11/pdf/04101.SWM2.Paper.pdf.

Mens, T., Van Gorp, P. (2005). *A Taxonomy of Model Transformation*. Elsevier B.V. Retrieved October 28, 2012, from http://staffwww.dcs.shef.ac.uk/people/ A.Simons/remodel/papers/MensVanGorpTaxonomy.pdf.

Mitchell, W.H. (2003). *UML the Unified Modeling Language*. Retrieved October 27, 2012, from http://www.mitchellsoftwareengineering.com/IntroToUML.pdf.

Octavian, P. (2003). *YATL Yet Another Transformation Language*. Oxford University Press: Oxford. Retrieved October 30, 2012, from http://kar.kent.ac.uk/14215/1/YATL_Yet_Another.pdf.

Oldevik, J. (2003). *Supporting Model-to-Model Transformations: The VMT Approach*.

Olteanu, D. (n.d). *XSLT 1.0 Tutorial*. Retrieved October 30, 2012, from http://www.cs.ox.ac.uk/dan.olteanu/tutorials/xslt1.pdf.

Patrascoiu, O. (n.d.). *Yet Another Transformation Language*. United Kingdom: University of Kent.

Peltier, M., Bezivin, J., and Guillaume, G. (n.d.).*MTRANS: A general framework, based on XSLT, for model transformations*.

Pure Systems.(n.d.).*IBM Rational Rhaspody*. Retrieved November 2, 2012, from http://www.pure-systems.com/uploads/media/pure_variants_ for_IBM_Rational_Rhapsody_-_2010-10.pdf.

Quartel, D., Dijkman, R., and Sinderen, M. (n.d.). *Extending profiles with stereotypes for composite concepts*. Retrieved November 6, 2012, http://doc.utwente.nl/63473/1/Quartel05extending.pdf.

Rahmouni, M. and Mbarki, S. (2011). *MDA-based ATL Transformation to Generate MVC 2 Web Models*, IbnTofail University. Retrieved October 26, 2012, from http://www.airccse.org/journal/jcsit/0811csit05.pdf.

Rational Software Corporation. (2000). *The UML and Data Modeling*, Cupertino, CA: Rational Software Corporation. Retrieved October, 28, 2012, from http://www.uml.org.cn/oobject/tp180.pdf.

Sendall, S., Perrouin, G., Guelfi, N., and Birberstein, O.(n.d.). *Supporting Model to Model Transformations: The VMT Approach*. Biel: Biel School of Engineering and Architecture.

Sostaks, A., Kalnina, E, Kalnins, A, Celms, E., and Iraids, J. (2011). *Solving the TTC 2011 Reengineering Case with MOLA and High-Order Transformations*. University of Latvia. Retrieved October, 30, 2012, http://cgi.cse.unsw.edu.au/~rvg/eptcs/ Published/TTC2011/Papers/27/arXiv.pdf.

Stephan, M. and Stevenson, A. (n.d.). *A comparative look at model transformation languanges*.

Syriani, E. and Vangheluwe, H. (2009). Matters of model transformation. *School of Computer Science, McGill University*.

Taentzer, G. (n.d.). *Characterizing Tools for Visual Modeling Techniques*. Berlin: Technical University of Berlin.

Tamura, G. and Cleve, A. (2010). *A comparison of Model Transformation Languages*. Retrieved October 27, 2012, from http://hal.inria.fr/docs/00/48/87/65/PDF/model-transformation-languages.pdf.

Tilton, J. (2001). *Introduction to XSLT*. Retrieved October, 30, 2012, from http://web.princeton.edu/sites/isapps/jasig/2001WinterDestin/presentations/Tilton/im011202.pdf.

Transforming XML with XSLT. (n.d.). *Transforming XML with XSLT*. Retrieved November 7, 2012, from http://oreilly.com/catalog/orxmlapp/chapter/ch07.pdf.

Tratt, L. (2004). Model transformations and tool integration. Department of Computer Science, King's College London.

Tratt, L. (2008). A change propagating model transformation language.*Journal of Object Technology*, 7 (3), 108 – 126. Retrieved October 29, 2012, from http://www.jot.fm/issues/issue_2008_03/article3.pdf.

United Modeling Language (UML) Basics (n.d.). Retrieved October 27, 2012, from http://www.ghotul.com/G/modules/downloads/downloads/UML.pdf.

Villard, L. and Layaida, N. (2002). *An Incremental XSLT Transformation Processor for XML Document Manipulation*. Retrieved October 28, 2012, from http://xsltinc.berlios.de/docs-extra/incXSLT.pdf.

Wagelaar, D., Straeten, R.V.D., and Deridder, D. (2010). *Module superimposition: a composition technique for rule-based model transformation languages*, Retrieved October 27, 2012, from Ebscohost database (DOI 10.1007/s10270-009-0134-3).

Wang, W. (2005).*Evaluation of UML Model Transformation Tools*. Business Informatics Group. Retrieved November 9, 2012, from http://www.big.tuwien.ac.at/system/theses/61/papers.pdf?1298558494.

Willink, E.D. (2003). *UMLX:* A Graphical Transformation Language for MDA.*2nd Generative Techniques in the Context of the Model Driven Architecture Workshop*.

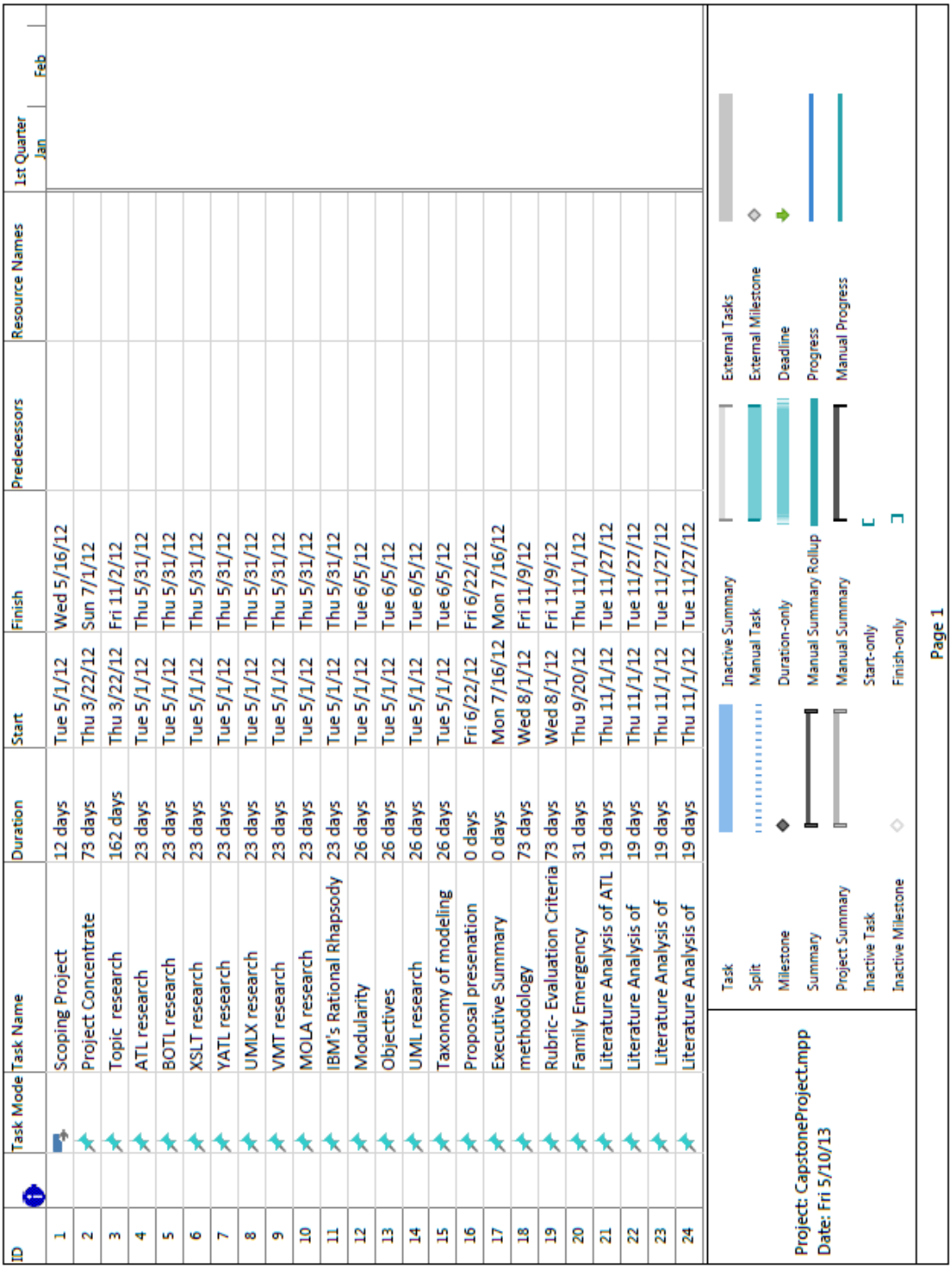
Willink, ED. (2003). *UMLX: A graphical transformation language for MDA*. Laboratory, University of Kent, United KingdomO.Patrascoiu@kent.ac.uk.

Appendix

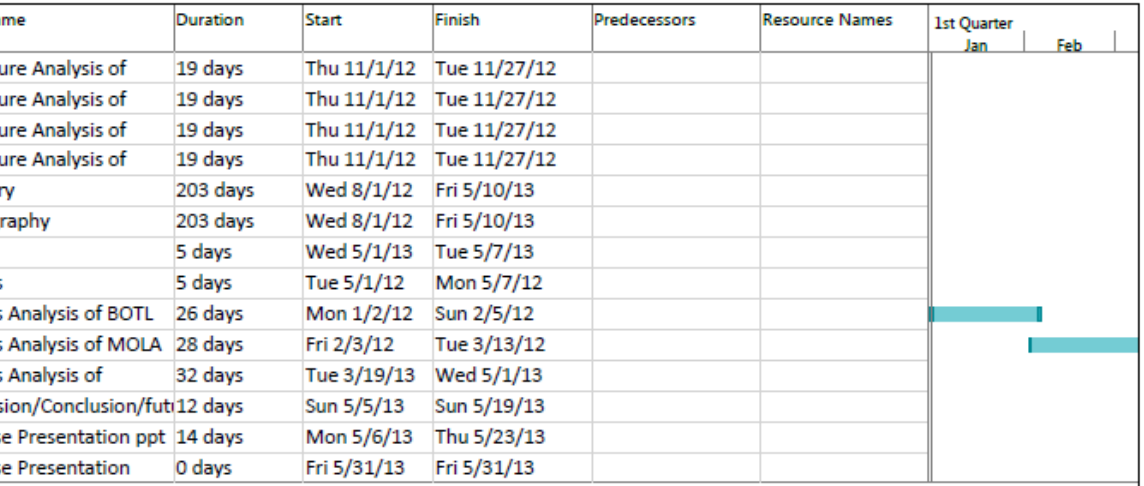
A. MS Project Timeline

MS Project Timeline

A Gantt Chart exhibiting tasks that were performed to complete this project.



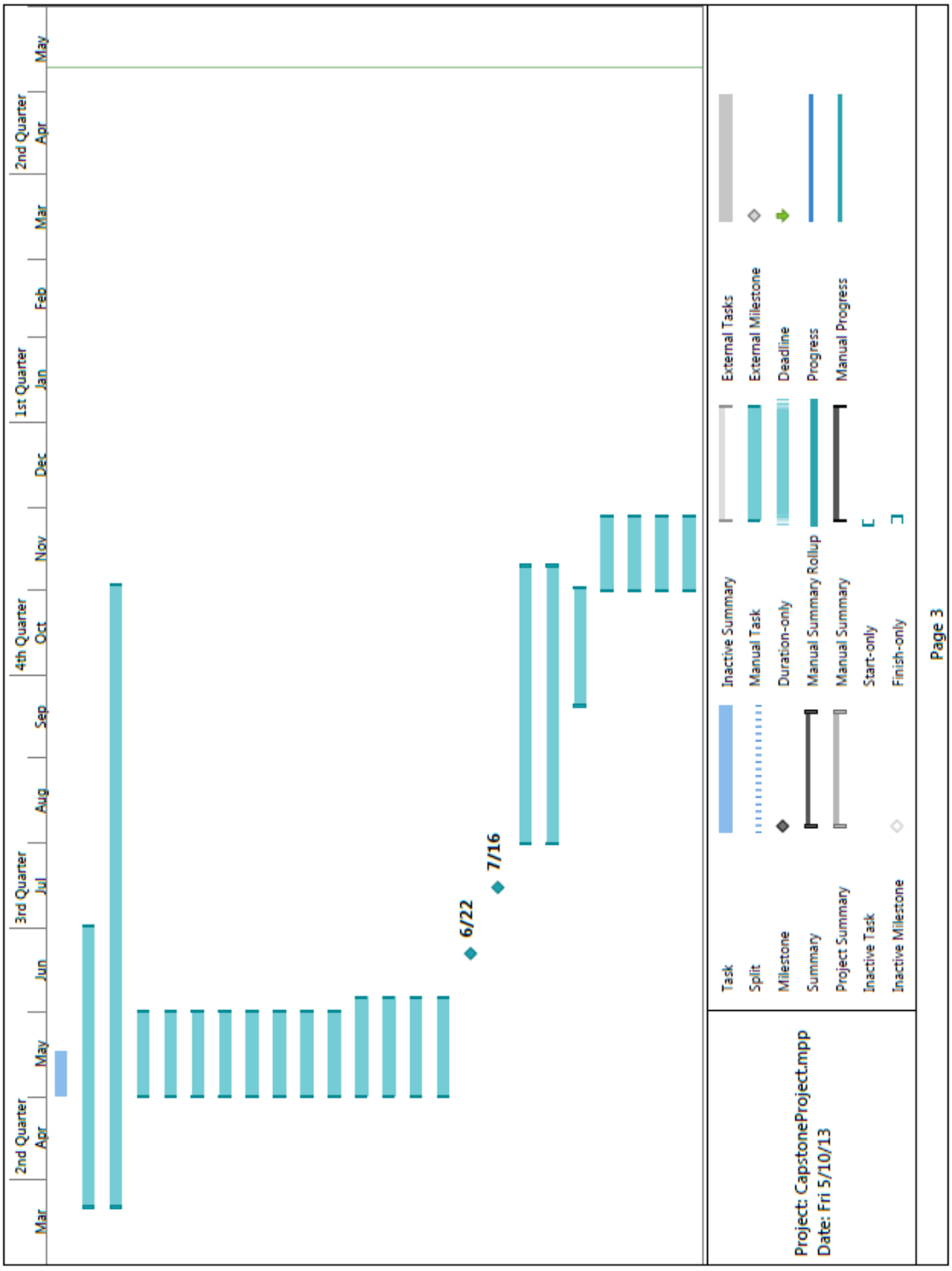
ID	Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
25		Literature Analysis of	19 days	Thu 11/1/12	Tue 11/27/12		
26		Literature Analysis of	19 days	Thu 11/1/12	Tue 11/27/12		
27		Literature Analysis of	19 days	Thu 11/1/12	Tue 11/27/12		
28		Literature Analysis of	19 days	Thu 11/1/12	Tue 11/27/12		
29		Glossary	203 days	Wed 8/1/12	Fri 5/10/13		
30		Bibliography	203 days	Wed 8/1/12	Fri 5/10/13		
31		Tables	5 days	Wed 5/1/13	Tue 5/7/13		
32		Figures	5 days	Tue 5/1/12	Mon 5/7/12		
33		Results Analysis of BOTL	26 days	Mon 1/2/12	Sun 2/5/12		
34		Results Analysis of MOLA	28 days	Fri 2/3/12	Tue 3/13/12		
35		Results Analysis of	32 days	Tue 3/19/13	Wed 5/1/13		
36		Discussion/Conclusion/fut	12 days	Sun 5/5/13	Sun 5/19/13		
37		Defense Presentation ppt	14 days	Mon 5/6/13	Thu 5/23/13		
38		Defense Presentation	0 days	Fri 5/31/13	Fri 5/31/13		

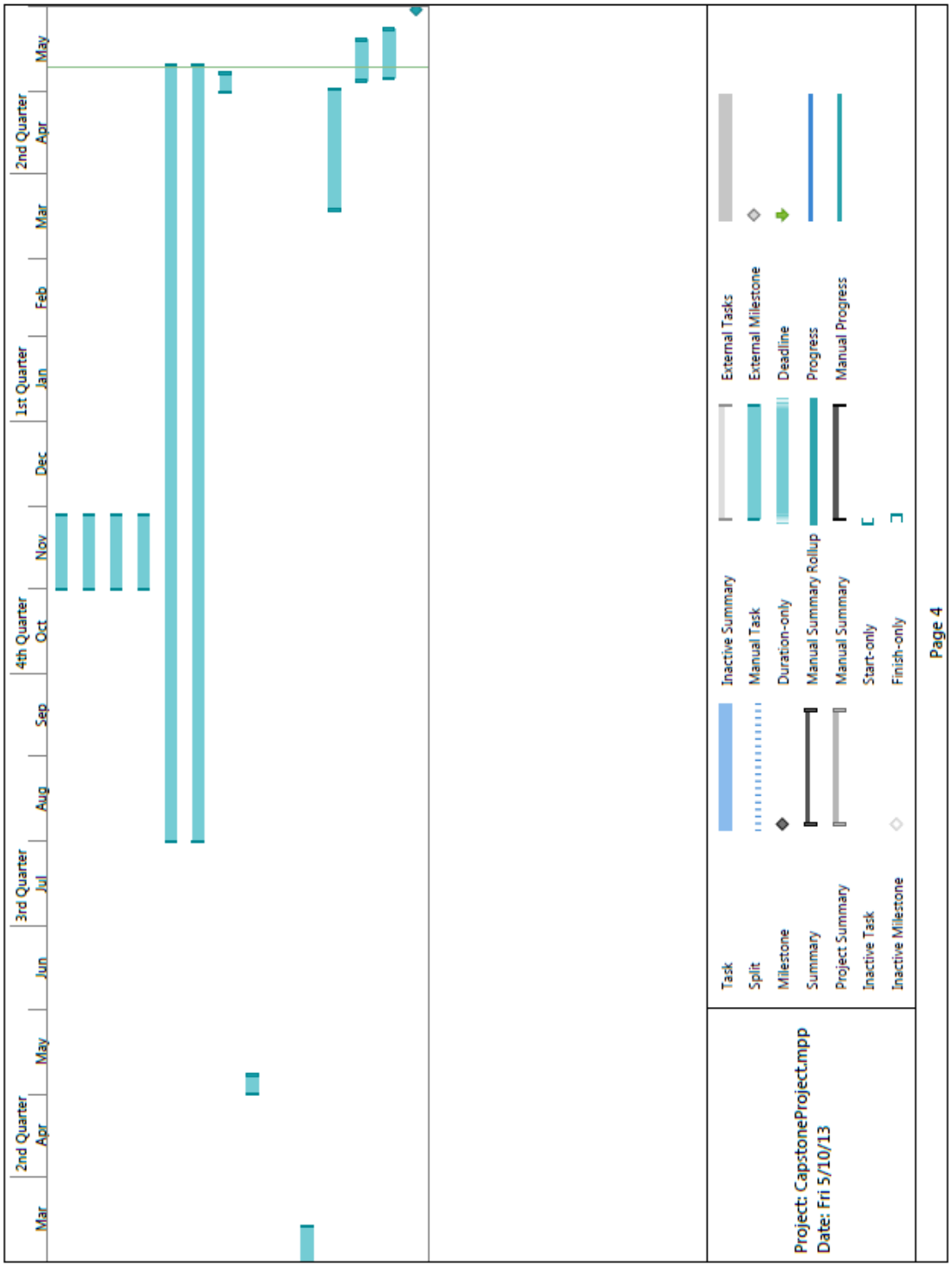


Project: CapstoneProject.mpp
Date: Fri 5/10/13

Task
Split
Milestone
Summary
Project Summary
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
External Tasks
External Milestone
Deadline
Progress
Manual Progress

Page 2


Mar
2nd Quarter
Apr
May
Jun
3rd Quarter
Jul
Aug
Sep
4th Quarter
Oct
Nov
Dec
1st Quarter
Jan
Feb
Mar
2nd Quarter
Apr
May
6/22
7/16
Project: CapstoneProject.mpp
Date: Fri 5/10/13
Task
Split
Milestone
Summary
Project Summary
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
External Tasks
External Milestone
Deadline
Progress
Manual Progress
Page 3


Mar
2nd Quarter
Apr
May
Jun
3rd Quarter
Jul
Aug
Sep
4th Quarter
Oct
Nov
Dec
1st Quarter
Jan
Feb
Mar
2nd Quarter
Apr
May
Project: CapstoneProject.mpp
Date: Fri 5/10/13
Task
Split
Milestone
Summary
Project Summary
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
External Tasks
External Milestone
Deadline
Progress
Manual Progress
Page 4

Appendix

B. Glossary

Appendix B

Glossary

- **Activity diagram** – Shows the flow of activity within a system.
- **Atlas Transformation Language (ATL)** – Is a modeling language and a tool in the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models.
- **Behavior Diagram** – Shows how the system is supposed to interact with other systems, users or objects within the program itself (i.e. Activity diagram).
- **Bijective** – one to one mapping.
- **Boolean expression –**evaluates to a boolean state usually represented as two possible states.
- **BOTL** – Bi-Directional Object Oriented transformation Language A mathematical transformation language and tool that can be used to specify transformations on a single model.
- **Class diagram** – Shows the static design view of a system, including packages, classes, interfaces, collaborations and their relationships.
- **Collaboration diagram** – Shows the interaction among objects but emphasizes the structural organization of the objects that send and receive information.
- **Component diagram** – Shows the static implementation view of a system.
- **Declarative Programming** – the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.
- **Deployment diagram** – Shows the connectivity of physical nodes in an architectural view of the system.
- **Directionality** – direction in which a rule may be executed or a model may be used; Bi-directional or uni-directional.
- **ECORE** – The Ecore Tools component provides a complete environment to create, edit and maintain Ecore models. This component eases handling of Ecore models with a Graphical Ecore Editor and bridges to other existing Ecore tools (Validation, Search, Compare, Emfatic, generators...). The Graphical Ecore Editor implements multi-diagram support, a custom tabbed properties view, validation feedbacks, refactoring capabilities. The long-term goal is to provide the same level of services as does JDT for Java.
- **EMFT** – Eclipse Modeling Framework Technology - The Eclipse Modeling Framework Technology project exists to incubate new technologies that extend or complement EMF.
- **Endogenous** – Action or object coming from within the system (i.e. using the same transformation language).
- **Exogenous** – Action or object coming from outside the system. (i.e. Using differently transformation languages).
- **Flow chart** – Very similar to an Activity Diagram, in that shows the flow of activity within a system. Does not have the notation to show parallel processes like with an Activity Diagram.
- **Formalism** – refers to the set of rules and symantics in the programming language.
- **Graph transformation** – Creating a new graph(s) from an original graph.
- **Graphical notation** – use of words to convey information. ISO/IEC 24744 provides an agreed-upon set of words (a vocabulary), plus their corresponding meanings (their semantics), that can be used to describe methodologies used to develop software, hardware and other similar products.
- **IDE** – Integrated Development Environment - consists of source code editor, build tool, and debugger.
- **Imperative** – In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state.
- **Lexical notation** – Short hand syntax for programmer's code and notes (e.g. Foo , meaning alphanumeric string of characters).
- **MDD** – Model Driven Development. MDD and MDE are sometimes used interchangeable. MDD is a software engineering approach that uses models to create software systems/applications.

- **Model** – A model is a artifact that indicates how something in the real world can be constructed.
- **Model Driven Architecture (MDA)** – A software design approach for the development of software systems. Object Management Group **(OMG®)**Model Driven Architecture (MDA) provides an open, vendor-neutral approach to the challenge of business and technology change. Based on OMG's established standard, the MDA separates business and application logic from underlying platform technology. Platform independent models of an application or integrated systems business functionality and behavior using OMG modeling standards can be used on virtually any platform, open or proprietary.
- **Model Driven Engineering (MDE)** – A methodology which uses models to express system details in domain models.
- **Modularity** – The degree to which a program can be taken apart and pieces can be used in other programs.
- **MOF** – Meta Object Facility -OMG® standard for MDE.
- **Multiplicity** – using more than one model. E.g. going from a PIM to several PSMs.
- **Object Management Group (OMG®)** – http://www.omg.org/ an international, open membership, not-for-profit computer industry standards consortium. Founded in 1989.
- **Object model diagram** – Shows static snapshots of instances of things found in the class diagrams.
- **Object oriented approach** – This is when data is stored within a method. Each method has a particular process. Methods are then used to talk with each other versus having direct access to the data.
- **OCL** – Object Constraint Language - a declarative language for describing rules that apply to UML models.
- **Operational** – taking 2 conflicting states and making them compatible, to go to 1 resolved state. Used in consistency checking and and collaboration..
- **Operational approaches** – use augmented metamodels with imperative constructs .
- **Package diagram** – Shows how a system is split up into logical groupings and their dependencies.
- **PIM** – Platform Independent Model A model at the highest level is independent of any implementation technology.
- **PSM** – Platform Specific Model. A model that is linked to a specific technological platform.
- **QVT** – Query, View, Transformation. A standard set of specifications for languages for model transformation defined by OMG®.
- **Refactor** – to run the program multiple times in order to evolve/refine it.
- **Reverse engineering** – To be able to take a fully coded program and create documentation for it (i.e. creating of the Class Diagram from the code). In reverse engineering you take a product at any point and go backwards figuring out how it was built from square one.
- **Schema** – A way to define the elements of a system. Schematics are ways to express diagrams.
- **Sequence diagram** – Shows the interaction among objects/users by emphasizing the order in which events take place in the system.
- **State Diagram** – Shows the states, transitions, events and activities within a system.
- **Structure diagram** – Shows things that must be present within the system (i.e. Class or Object diagrams).
- **Structure driven approach** – runs the transformation in phases .
- **Taxonomy** – Arrangement, groups/groupings of objects with similar traits.
- **UML** -Unified Model Language – In software engineering is a standardized modeling language which includes a set of graphic notations to create visual models.
- **UMT** – UML Model Transformation Tool.
- **Use case diagram** – Shows the behavior of a system, subsystem or a class.
- **VMT** – Visual Model Transformation - Is a language and transformation tool.
- **XSL** – eXtensible Style Sheet Language.
- **XSLT** – XSL tool for Transformation.

Appendix

C. An Emperical Study of UML Model Transformation Tools (UMT)

An Empirical Study of UML Model Transformation Tools (UMT)

Requirements	MOLA	BOTL	VMT	YATL	XSLT	ATL	UMLX	Rational Rhapsody	Discussion
Inheritance	No	No	Yes	No	No	Yes	No	Yes	Rhapsody has ability to make a quicker new transformation as well as making changes relevant to many transformations.
Graphical notation	Yes	Yes	Yes	No	No	No	Yes	Yes	Rhapsody has ability specify transformation Graphical. The model also use UML-to-UML transformations to make graphical models. Typically, the graphical models delivers a higher-level view, which enhances communicate than the lexical counterpart.
Lexical notation	No	No	Yes	Yes	Yes	Yes	No	Yes	Rhapsody is also a language that could specify the transformation lexically. It has ability to handle a complex graphical notations having a scalability problem.
Declarative	No	Yes	No	No	Yes	No	Yes	Yes	Rational Rhapsody is able to support statements the transformation shall do. The importance of declarative is that it does not operate on a computer system. Moreover, declarative does not allow a programmer to alter the value of a declaration and a variable. The features make the declarative language to be side-effect free and making errors in the written code to be reduced.
Bidirectional	No	Yes	No	No	No	No	No	Yes	Rational Rhapsody is to support bio-directional language by supporting transformation in two-ways. The advantage is that bio-directional language is easier to maintain and specification is easier.

Requirements	MOLA	BOTL	VMT	YATL	XSLT	ATL	UMLX	Rational Rhapsody	Discussion
XML support	No	No	No	No	Yes	No	No	Yes	Rhapsody is also a XML support by providing a built-in support to produce or consume XML, which include ability to support general XML, attributes, elements, and namespaces. This requirement is very important for the text-to-UML and UML to-text due to widespread usage of data formats and XML specifications.
Text-to-XML	Yes	No	No	No	Yes	No	No	Yes	With reference to Text-to-UML, Rational Rhapsody has ability to specify reverse-engineering transformations from a code/text to UML models. Using this language, a graphical UML models could improve documentation and a model-driven setting.
UML-to-UML	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Moreover, the Rational Rhapsody has ability to transfer transformations from a UML model into another UML model. The benefit derived from this feature is to transform platform independent models into platform-specific models.
UML-to-TEXT	No	No	Yes	No	Yes	Yes	No	Yes	With the UML-to-TEXT language, Rational Rhapsody has ability to specify transformations from a UML model to text format. The benefit derived from this feature is to achieve a code generation and use the feature for a documentation purpose.
UML tool Independence	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Rational Rhapsody could use UML tool independence to support transformation of models into several UML tools. This requirement is desirable top derive a solution to a single UML tool.

Requirements	MOLA	BOTL	VMT	YATL	XSLT	ATL	UMLX	Rational Rhapsody	Discussion
No Proprietary Intermediate Structures	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Rational Rhapsody has been able to use this feature to reduce a complexity for a transformation architect.
Traceability	No	No	Yes	Yes	No	No	Yes	Yes	Rational Rhapsody ability to provide explicit traces on every target element. The feature assists a developer to understand the transformation, the source model and modification of the transformation to get better results from software development system.
MOF-based /Metamodel	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Rational Rhapsody uses this language to get better results from the transformation specification. Using the Metamodel-based language, a developer could be able achieve a correct transformation as well as checking whether the metamodel follows the explicit transformation rule.

From: Grønmo, R. (2001). An empirical study of the UML model transformation tool (UMT). SINTEF: Norway. Retrieved February 20, 2013, from http://heim.ifi.uio.no/~roygr/INTEROP-ESA-2005.pdf.
AND appended with a study of Rational Rhapsody Developer for C ++ by Adalia C. Hiltebeitel.

Appendix

D. Birdcage in Rational Rhapsody Developer

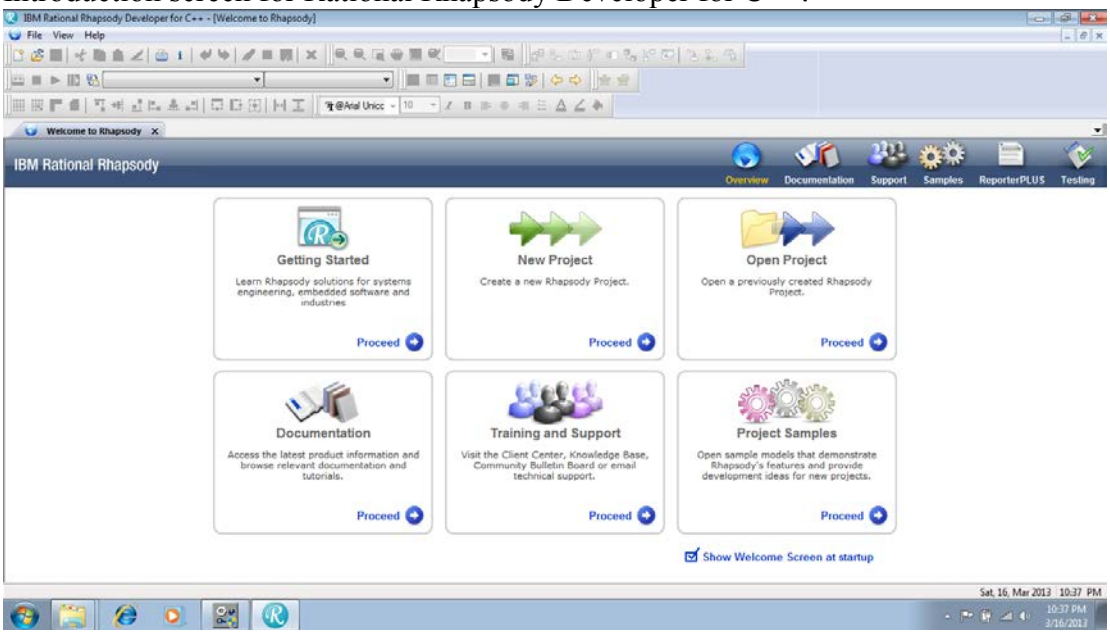
Birdcage in Rational Rhapsody Developer

The following is a Software engineering project. The goal is to use Rational Rhapsody to convert UML models into skeletal code. The program description is as follows:

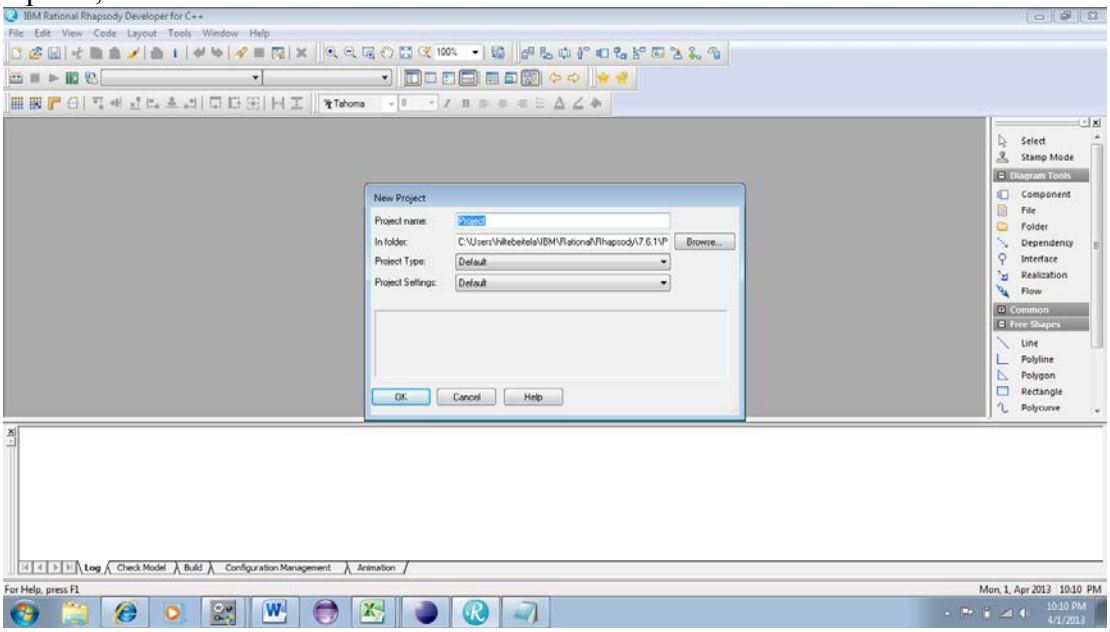
> The BirdCage Corporation needs software to aid the management of its business. The corporation builds its business around domestic birds. BirdCage arranges its business around different services with each service being managed by a different company making BirdCage a group of companies. Companies are divided into departments. Services include veterinary services, safekeeping of birds, bird calisthenics and a bird competitions service. In order to streamline its business, BirdCage classifies the veterinary and safekeeping services as medical services. The other two services are regarded as entertainment services. In the future the company hopes to add a number of "Nutritional" services to address the production and sales of food for birds. The corporation classifies birds as either exotic or non-exotic. BirdCage does not provide entertainment services for exotic birds. Owls and crows are non-exotic birds, while macaws, cockatoos and green quakers are exotic. BirdCage seeks to maintain a minimum stock of at least 10 of each type of exotic bird.

I have created the use case, class and component diagrams for the Birdcage Project. When installing The Rational Rhapsody Developer program it is very important to enure that one's telnet is enabled to allow the software to communicate with the licenses server for authorization.

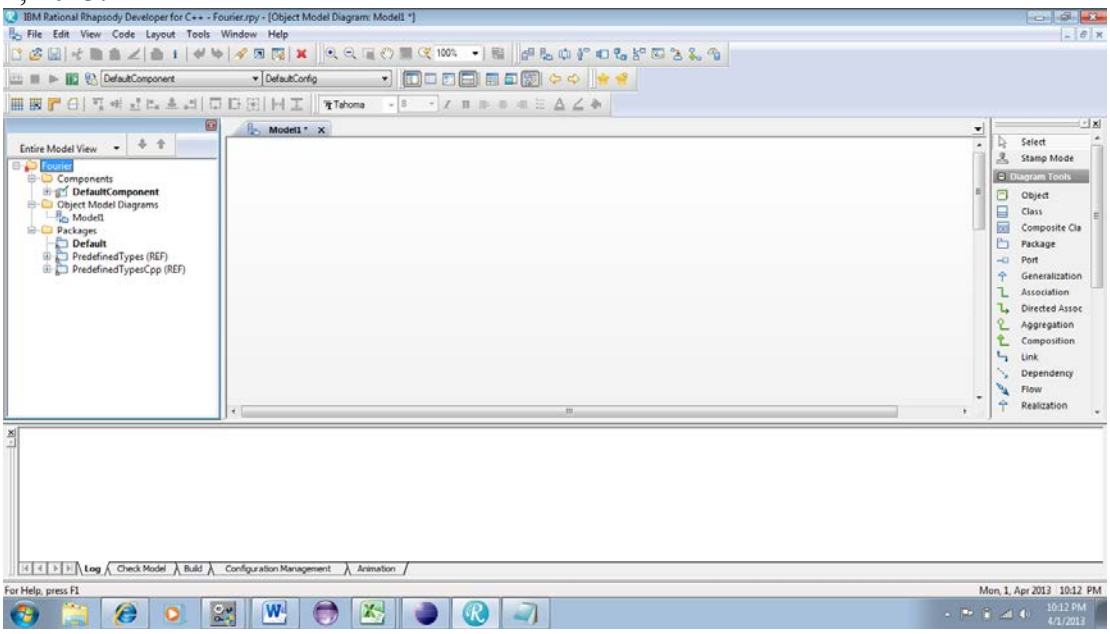
Introduction screen for Rational Rhapsody Developer for C++:



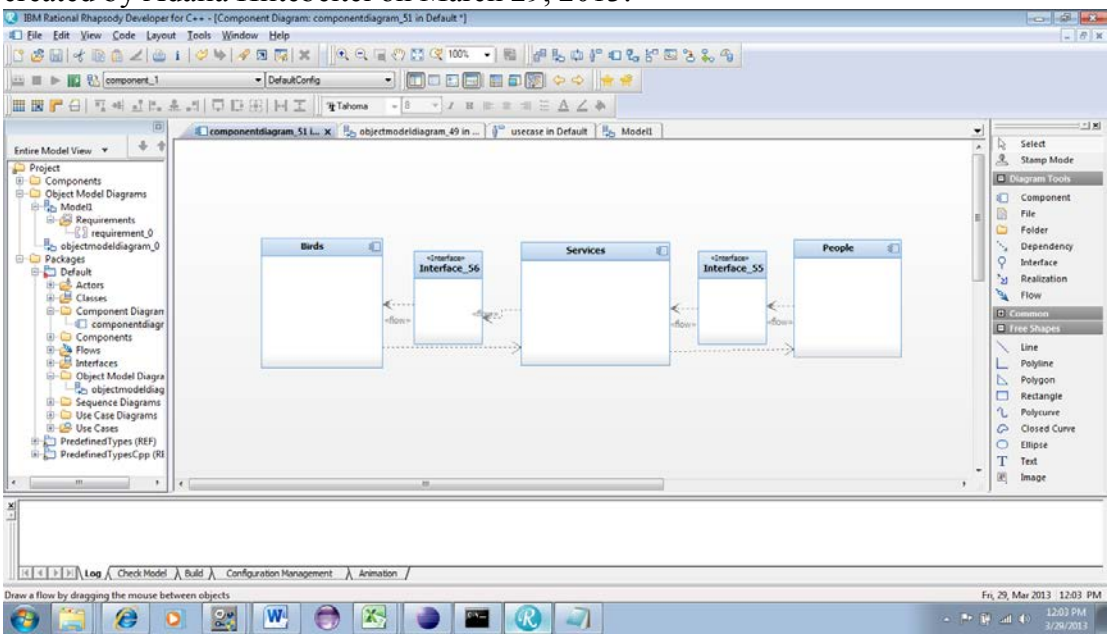
In this screenshot, the new project Birdcage is being created by Adalia Hiltebeitel on April 1, 2013:



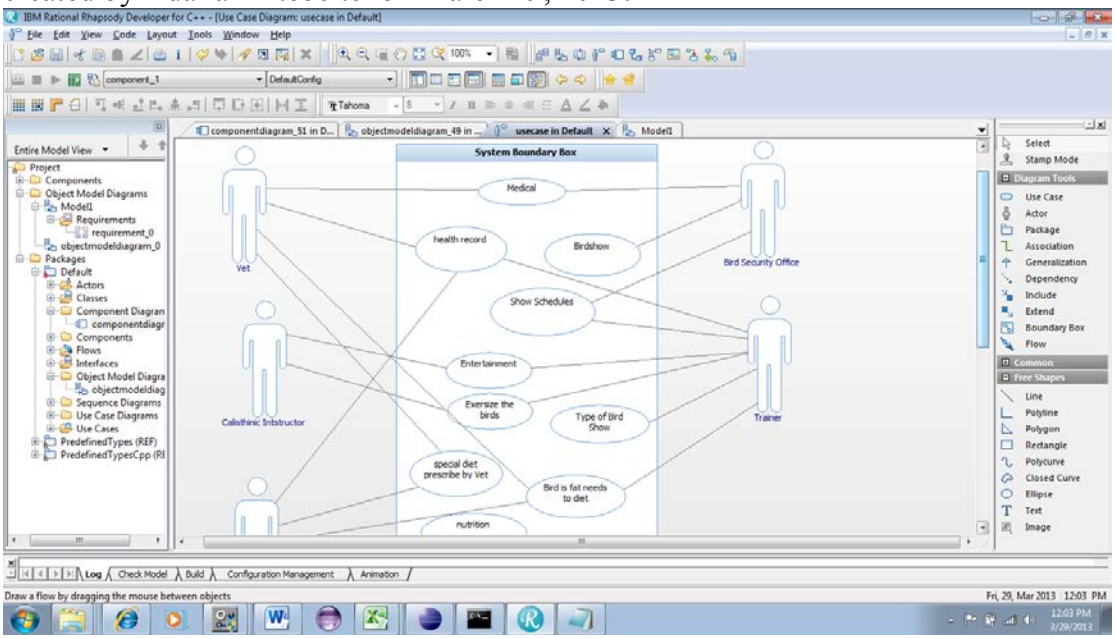
Blank template for building modeling diagrams created by Adalia Hiltebeitel on April 1, 2013:



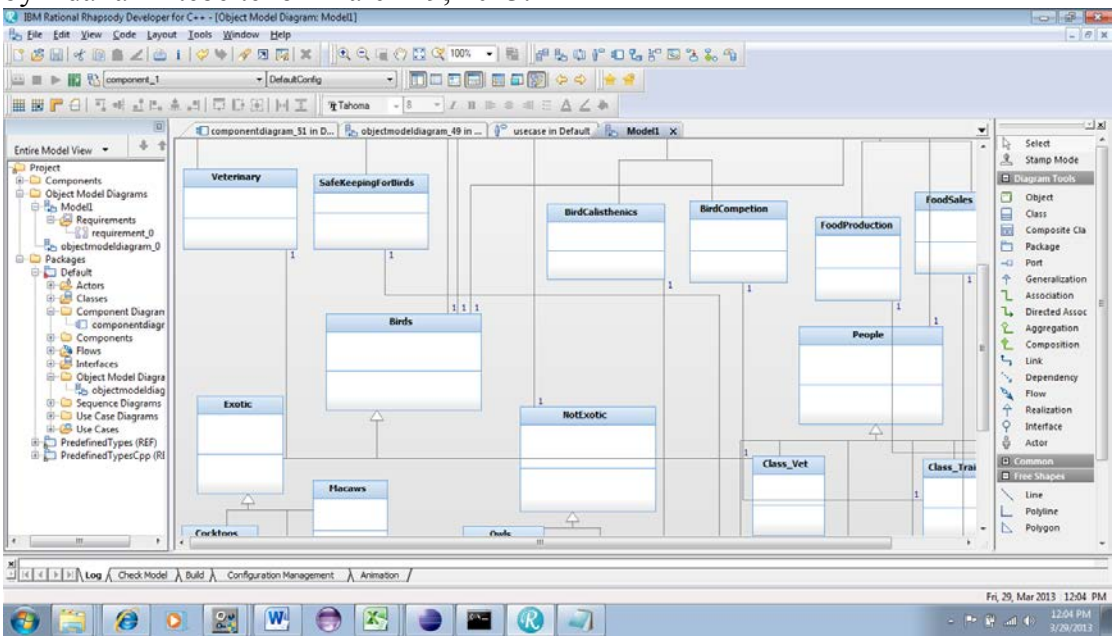
Component Diagram for Birdcage created in Rational Rhaspody Developer for C++ created by Adalia Hiltebeitel on March 29, 2013:



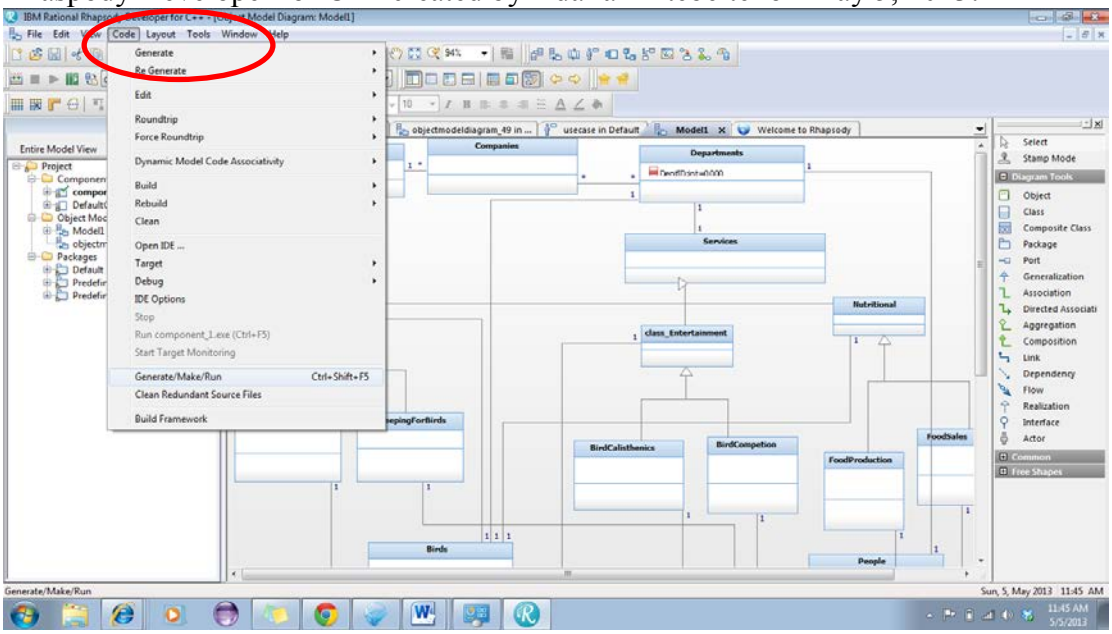
Use Case Diagram for Birdcage created in Rational Rhaspody Developer for C++ created by Adalia Hiltebeitel on March 29, 2013:



Class Diagram for Birdcage created in Rational Rhaspody Developer for C++ created by Adalia Hiltebeitel on March 29, 2013:



Generating code takes place under the Code tab. This was created in Rational Rhaspody Developer for C++ created by Adalia Hiltebeitel on May 5, 2013:



When your models have not been created accurately, you will receive Errors in the Log table that guide you to the corrections needed. This was created in Rational Rhaspody Developer for C++ created by Adalia Hiltebeitel on May 5, 2013:

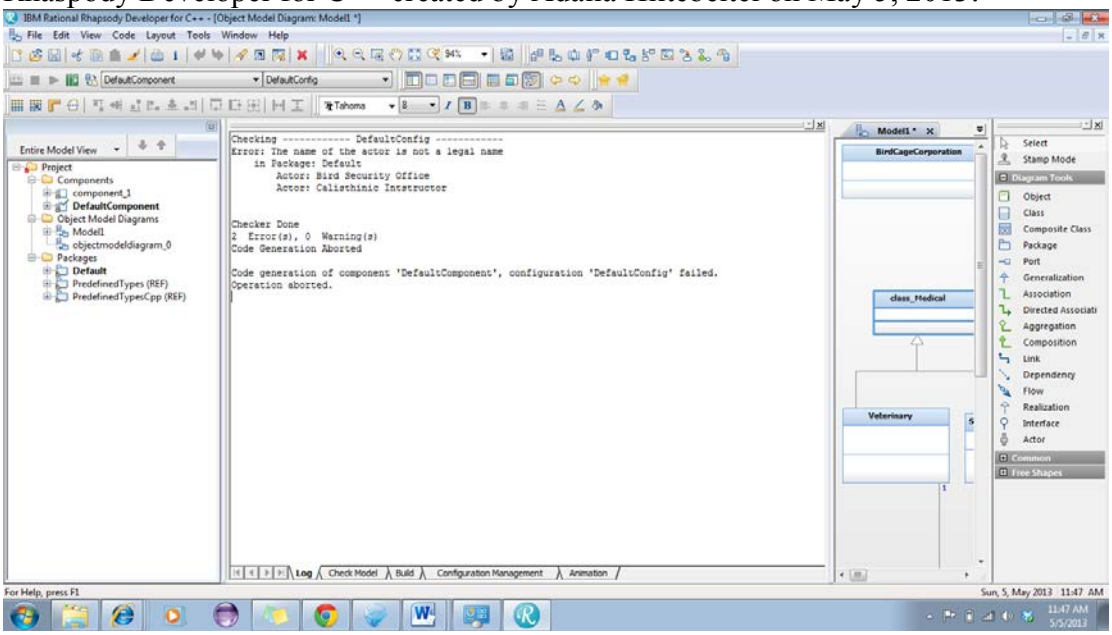


Diagram 1:

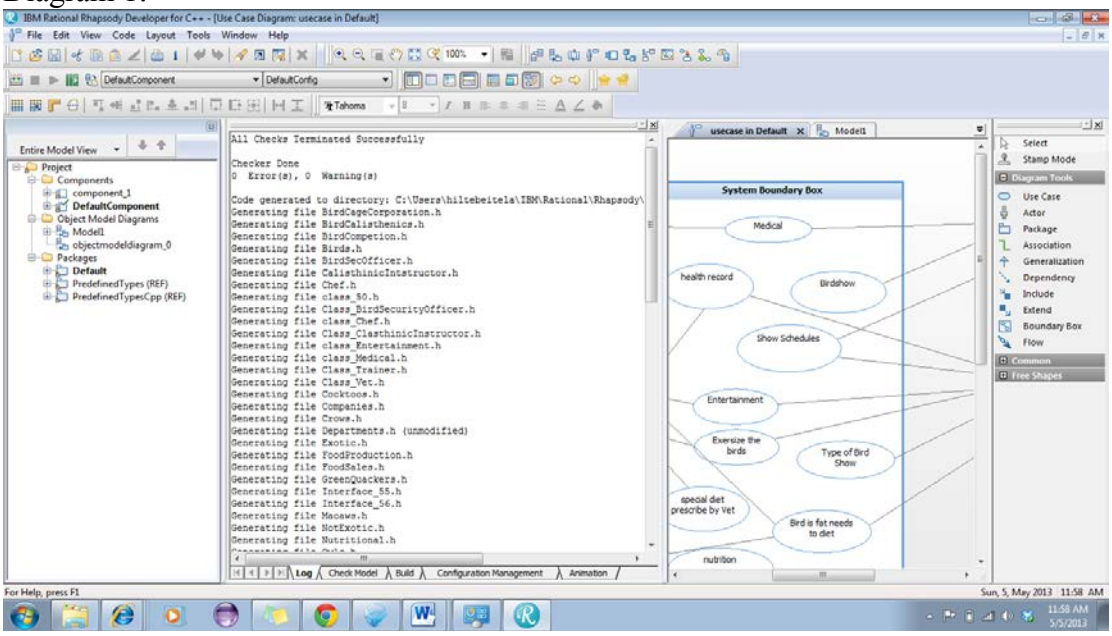


Diagram 2:

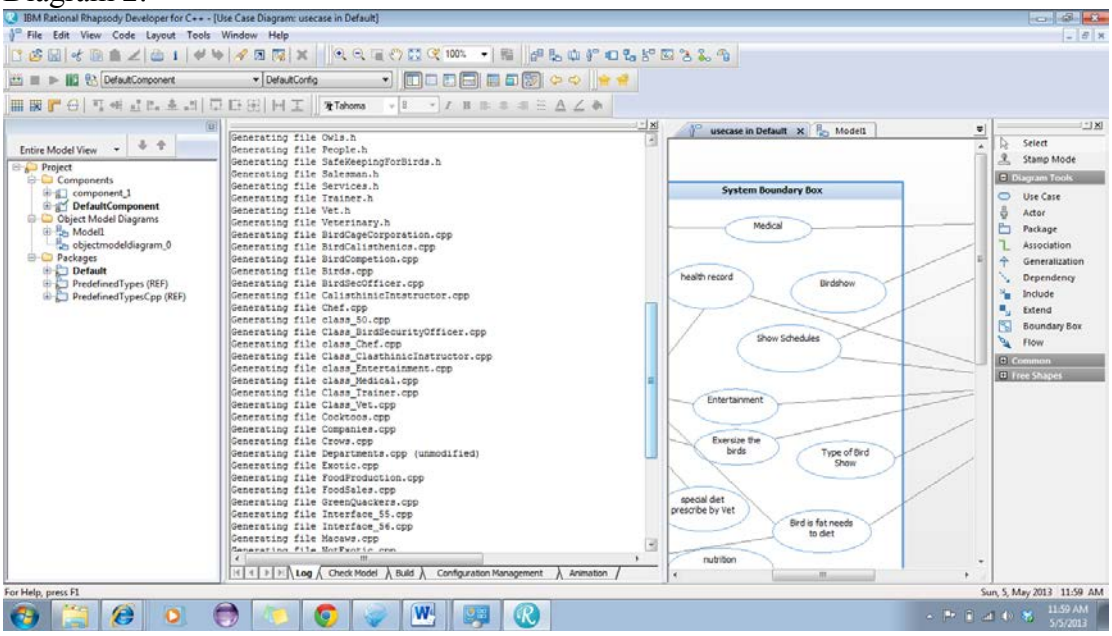
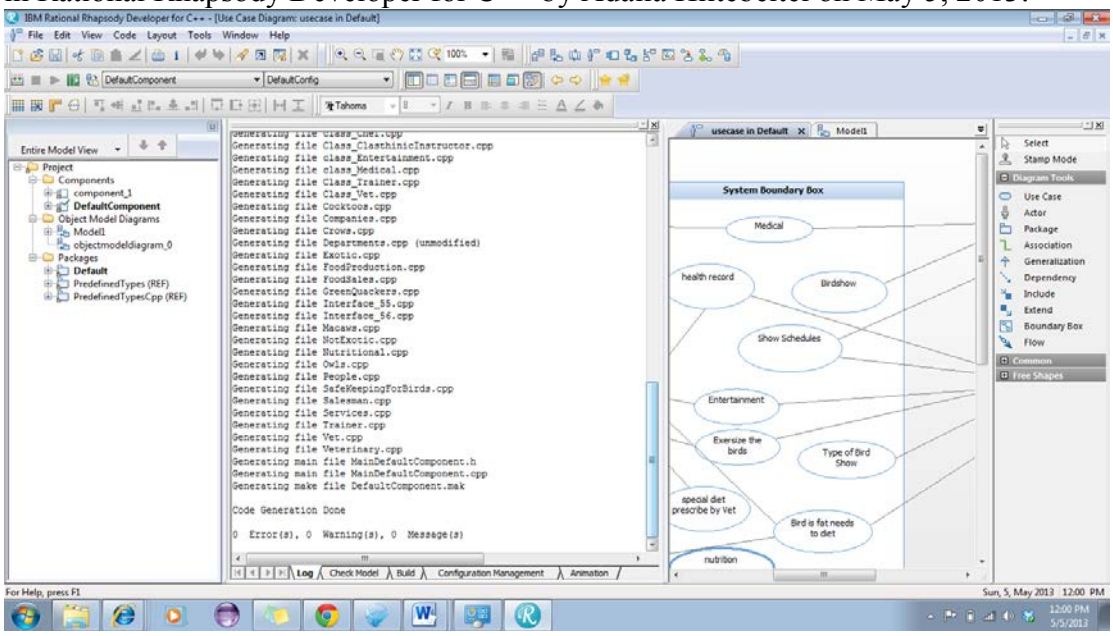
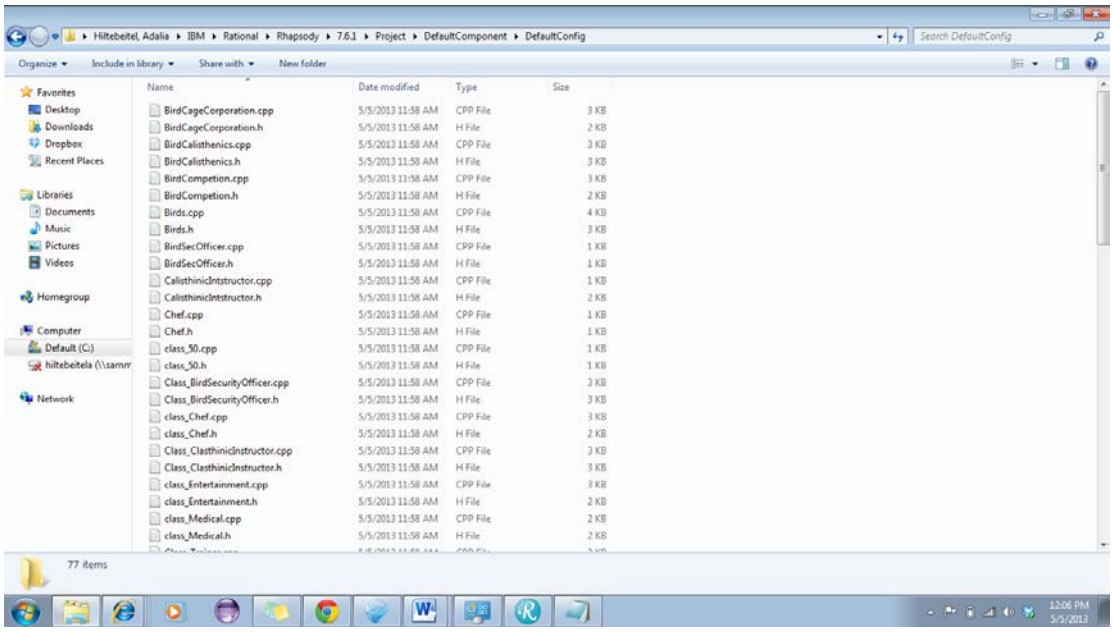


Diagram 3.

Once the errors have been corrected one is able to regenerate the project. The log file created a directory with text files shown in diagrams 1, 2, 3. The text files created are proofs the UML model to TEXT format feature does work for code generation, as well as being able to use this feature for documentation purposes. Files were created in Rational Rhapsody Developer for C++ by Adalia Hiltebeitel on May 5, 2013:





Rational Rhapsody text files have been created from the Birdcage UML models in the following location C:\Users\hiltebeitela\IBM\Rational\Rhapsody\7.6.1\Project\DefaultComponent\DefaultConfig.

An example of code generated by Rational Rhapsody, BirdCageCorporation text file.

```
/*********************************************************************
        Rhapsody        : 7.6.1
        Login           : hiltebeitela
        Component       : DefaultComponent
        Configuration   : DefaultConfig
        Model Element           : BirdCageCorporation
//!     Generated Date          : Sun, 5, May 2013
        File Path       : DefaultComponent\DefaultConfig\BirdCageCorporation.cpp
*********************************************************************
/

//## auto_generated
#include "BirdCageCorporation.h"
//## link itsCompanies
#include "Companies.h"
//## package Default

//## class BirdCageCorporation
BirdCageCorporation::BirdCageCorporation() {
}

BirdCageCorporation::~BirdCageCorporation() {
 cleanUpRelations();
}

OMIterator<Companies*> BirdCageCorporation::getItsCompanies() const {
 OMIterator<Companies*> iter(itsCompanies);
 return iter;
}

void BirdCageCorporation::addItsCompanies(Companies* p_Companies) {
 if(p_Companies != NULL)
 {
 p_Companies->_setItsBirdCageCorporation(this);
 }
 _addItsCompanies(p_Companies);
}

void BirdCageCorporation::removeItsCompanies(Companies* p_Companies) {
 if(p_Companies != NULL)
 {
 p_Companies->__setItsBirdCageCorporation(NULL);
 }
 _removeItsCompanies(p_Companies);
}
```

```
void BirdCageCorporation::clearItsCompanies() {
 OMIterator<Companies*> iter(itsCompanies);
 while (*iter){
 (*iter)->_clearItsBirdCageCorporation();
 iter++;
 }
 _clearItsCompanies();
}

void BirdCageCorporation::cleanUpRelations() {
 {
 OMIterator<Companies*> iter(itsCompanies);
 while (*iter){
 BirdCageCorporation* p_BirdCageCorporation = (*iter)-
>getItsBirdCageCorporation();
 if(p_BirdCageCorporation != NULL)
 {
 (*iter)->__setItsBirdCageCorporation(NULL);
 }
 iter++;
 }
 itsCompanies.removeAll();
 }
}

void BirdCageCorporation::_addItsCcompanies(Companies* p_Companies) {
 itsCompanies.add(p_Companies);
}

void BirdCageCorporation::_removeItsCompanies(Companies* p_Companies) {
 itsCompanies.remove(p_Companies);
}

void BirdCageCorporation::_clearItsCompanies() {
 itsCompanies.removeAll();
}

/*********************************************************************
*
        File Path        : DefaultComponent\DefaultConfig\BirdCageCorporation.cpp
*********************************************************************
/
```