Department of Information Science and Technology

A Systematic Comparison of Roundtrip Software Engineering Approaches

Dionisie Rosca

Dissertation submitted as partial fulfillment of the requirements for the degree of
Master's in Information Systems Management

Supervisor:
Luisa Domingues, PhD, Assistant Professor
ISCTE-IUL

December 2019

Acknowledgments

I would like to express my gratitude to my advisor, Professor Luisa Domingues, who believed in my abilities, gave me all the necessary support and was always available to help.

Finally, I would like to express my thanks to all those who were involved in my day to day to achieve this dissertation.

Abstract

Model-based software engineering contemplates several software development approaches in which models play an important role. One such approach is round-trip engineering. Very briefly, round-trip engineering is code generation from models, and models are updated whenever a code change occurs.

The objective of this dissertation is to benchmark the comparative analysis of the round-trip engineering capability of the UML, Papyrus, Modelio and Visual Paradigm modeling tools. In more detailed terms, the work will focus on evaluating tools to automatically or semi-automatically support round-trip engineering processes for each selected diagram. Collaterally, this dissertation will allow us to gain insight into the current round-trip engineering landscape, establishing the state-of-the-art UML modeling tool support for this approach.

Qualitative and quantitative analysis of the round-trip engineering capabilities of the tools show that the Papyrus, Modeling and Visual Paradigm tools yielded satisfactory results by applying the Reverse and Forward Engineering scenarios without changing the models and codes but applying the Round-trip engineering scenario with changes in model and code presented results with some gaps in model and code coherence. It was concluded that they arose because the semantic definition of the models was done informally. The conclusions drawn throughout the dissertation will answer the questions: How effective are current code generation tools for documenting application evolution? Where will it support the decision made? objectives and will support the recommendations of the best tools that address the round-trip engineering method.

Keywords: Model-driven engineering, round-trip Engineering, forward engineering, reverse engineering, UML modeling tools, metamodel, models, code generation, traceability, benchmarking

Resumo

A engenharia de software baseada em modelo contempla várias abordagens de desenvolvimento de software nas quais os modelos desempenham um papel importante. Uma dessas abordagens é a Round-trip engineering. Muito brevemente, a Round-trip engineering é a geração de código a partir de modelos, e os modelos são atualizado sempre que ocorre uma alteração no código.

O objetivo desta dissertação é a realização de um benchmarking da análise comparativa da capacidade de Round-trip engineering das ferramentas de modelação UML, Papyrus, Modelio e Visual Paradigm. Em termos mais detalhados, o trabalho se concentrará na avaliação de ferramentas para dar suporte automático ou semiautomático a processos de Round-trip engineering (engenharia direta e engenharia reversa) para cada diagrama selecionado. Colateralmente, esta dissertação permitirá alcançar uma visão do panorama atual da Round-trip engineering, estabelecendo o estado da arte do suporte de ferramentas de modelação em UML à dita abordagem.

A analise qualitativa e quantitativamente da capacidade de Round-trip engineering das ferramentas mostro que, as ferramentas Papiro, Modelagem e Paradigma Visual apresentaram resultados satisfatórios aplicando os cenários de Reverse e Forward Engineering sem alterar os modelos e códigos e com alterações, mas aplicando o cenário Round-trip engineering com alterações nos modelo e código apresentaram resultados com algumas lacunas nomeadamente na coerência dos modelos e código. Concluiu-se que as mesmas surgiram por causa da definição semântica dos modelos ser feita de forma informal. As conclusões tiradas ao longo do trabalho respondera as perguntas: Qual a eficácia das ferramentas atuais de geração de código para documentar a evolução dos aplicativos? Onde apoiará a decisão tomada? que foram definidas nos objetivos e apoiarão as recomendações das melhores ferramentas que aborda o método Round-trip engineering.

Palavras-Chave: Model-driven engineering, round-trip engineering, forward engineering, reverse engineering, ferramentas de modelação UML, metamodelo, transformação de modelos, geração de código, rastreabilidade

Contents

Tables

Figures

Abbreviations and Acronyms

BPM – Business Process Management

DSL – Domain-Specific Language

EMF – Eclipse Modeling Framework

GEF – Graphical Editing Framework

GMF – Graphical Modeling Framework)

MDA – Model-Driven Architecture

MDD – Model-Driven Development

MDE – Model-Driven Engineering

MOF – Meta-Object Facility

OCL – Object Constraint Language

OMG – Object Management Group

OO – Object-Oriented

RCP – Rich Client Platform

SOA – Service-Oriented Architecture

SVN – Subversion

SysML – System Modeling Language

TOGAF – The Open Group Architecture Framework

UML – Unified Model Language

XML – Extensible Markup Language

XSD – XML Schema Definition

WSDL – Web Services Description Language

Chapter 1 – Introduction

1.1. Scope

With this dissertation, it is intended to systematically and rigorously compare the roundtrip engineering capabilities of a selected set of modeling tools: Papyrus, Modelio and Visual Paradigm, which will result in benchmarking of comparative analysis of the capabilities of tools. Collaterally, this dissertation will provide an overview of the current roundtrip engineering landscape, establishing the state of the art of UML (Unified Modeling Language) modeling tool support for this approach. The UML tools will be evaluated qualitatively and quantitatively, duly selected according to specific and well-founded criteria.

In more detail, it will focus on evaluating the selected tools in their automatic or semi-automatic support of round-trip engineering subprocesses (forward engineering and reverse engineering) for each overall selected diagram. Since the same evaluation method will also be applied to each involved UML metamodel modeling element, not all diagrams with possible transformation into source code will be studied.

It is intended to evaluate the relationship of the diagrams / modeling elements with the constructs of at least one given object-oriented programming language. A detailed understanding of the traceability mechanism required for synchronization between artifacts located at different levels of abstraction (models and source code) in each tool will be achieved, which depends on model transformations.

From a quantitative point of view, the objective of this dissertation is to determine, in terms of quantitative comparison of indicators, the coverage of each tool regarding the generation of source code from a certain set of UML (forward engineering) modeling diagrams and elements. Each will also be covered with respect to changing diagrams and modeling elements in the presence and according to changes in the source code previously generated from these higher-level artifacts (reverse engineered).

1.2. Motivation

Software models are a fundamental tool for engineers to rationalize a given system without, however, relying on a high level of technological detail. The relationship between the complexity of the system and the importance of the models is directly proportional, being not only the complexity of contemporary systems ever greater, but

also the difficulty of understanding and maintaining complex systems in the face of the impossibility of using abstract representations of them.

The models are equally powerful about communication between the various stakeholders in system development, just as visual notation is more comfortably assimilated than pure code (Ardis, 2000), (Atkinson and Kühne, 2003).

The automatic or semiautomatic support of tools to models-oriented approaches, such as roundtrip engineering, allows to increase the consistency of this subprocess of software development, automating integration activities between phases of software development from analysis, through design until implementation, and therefore different types of model and source code. Thus, reducing the time and effort of the development and contributing to the maintenance of the quality of the system under construction. The important role of the introduction of the roundtrip engineering topics with UML in the teaching programs of undergraduate students is recognized, and to this end tools have been developed to support it, particularly in connection with the Java language, in the form of plugins for Eclipse (Model Goon, 2011), (Usman and Nadeem, 2009).

According to Khaled (2009) through the last few years, building systems became very complicated and any complicated system needs a tool to be built For this reason, research work (Khaled, 2009) and (Wang, 2001) has focused on the generic comparison of UML modeling tools, which means that there are no properly detailed scientific evaluations carried out over the past decade to support these tools. the round-trip engineering approaches. The benchmarking to be carried out in this dissertation will serve as a decision support for engineers involved in software development projects oriented to the models, namely the decision on which roundtrip engineering tool to adopt.

Additionally, with the proposal of a qualitative evaluation framework and the definition of metrics for quantitative evaluation, opportunities for improvement of the existing modeling tools in UML will be identified, in relation to roundtrip engineering support.

1.3. Research Questions and Goals

The issue in any development software project is how to produce the executable components from the requirements specification and how to support this in the long run process.

The object-oriented approach support for a change from one phase to another. However, change requests and problems occur more often in the implementation phase. This phase is not fully implemented in artifacts. Even more difficult it is when, the design artifacts are kept up-to-date with the deployment artifacts.

With UML and round-trip engineering, the programmers generate the source code from a UML model, modifying that code in a development environment, and recreates adapted UML model from the source code. The following approach promotes better time and better quality. This thesis studies the whole domain of the engineering problem, methodologies and round-trip tools.

It is intended, with this master's dissertation, to produce the state-of-the-art covering UML modeling tools to support roundtrip engineering, to have an updated overview on the evolution of the UML and the tools that are available in the market. Afterwards, a qualitative evaluation framework of this type of tools will be made, composed of rigorous evaluation criteria. Then metrics will be defined to cover the processes of forward engineering and reverse engineering, by each tool. The authors will propose recommendations for using a round-trip engineering tool and will attempt to answer the question: How effective are current code generation tools for documenting application evolution? Where will it support the decision?

1.4. Methodological Approach

In this chapter we will describe step by step the benchmarking method (Coffel, 2010) to be used in this dissertation that will be applied to a group of UML modeling tools. Figure 1 details the benchmarking process.

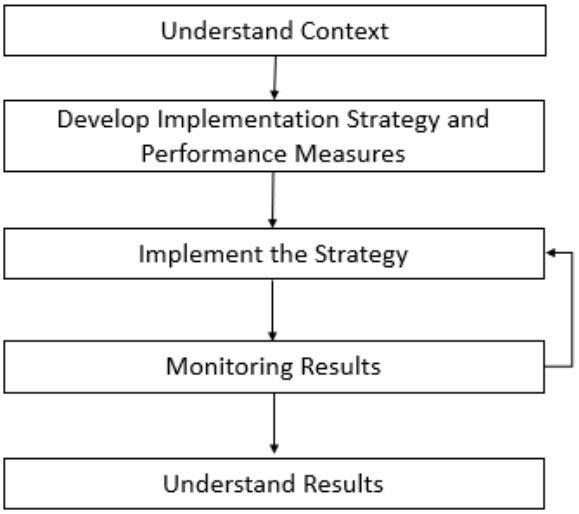

Figure 1 – Benchmarking Phases

In the first phase the context of benchmarking was defined, i.e. the method should be designed to ensure the systematic and rigorous nature of the assessment approach, using a repeatable case in both qualitative and quantitative assessment of all tools. The case should include diagrams and modeling elements to choose UML metamodel according to clear criteria. Using a systematic and rigorous approach will allow us to come to solid conclusions about the degree of support of each of the tools for the round-trip engineering approach. Model-based software engineering contemplates various software development strategies in which models play a central role. Round trip engineering is one such approach. Very briefly, it provides code generation of models and the model is updated whenever a change occurs in the code.

In the second phase, the develop implementation strategy and performance measures, which aims to define which approach will be applied in the case studies and will be defined what are the criteria used to compare the implemented case studies, based on the case studies of other authors. Since UML was standardized, the language has been extended to System Modeling Language (SysML) to solve system development problems. Programmers, business analysts, architects, systems analysts who trust the UML must learn the language. For them, UML diagrams are not simple images for communication purposes and documentation is a view in a formally different way (Booch, Rumbaugh, Jacobson, 2006).

In phase three, the implementation of strategy, will be conduct the experiments that consist of generating the code from UML diagrams, applying the Forward Engineering and the second case generating the UML diagrams from code, Reverse Engineering. In the third case we will try to synchronize the diagrams and the code, when each one is modified, that mean Round-trip Engineering. In this same phase will be collected data.

In the fourth phase, the results monitoring is where data analysis will be done for each tool and case, as well as a comparison between the tools and their cases, ensuring that all the proposed cases are covered. If during this phase inconsistencies is detected, step three will be repeated until satisfactory results.

In the fifth phase, understand results, is where conclusions will be drawn, that is, the results of the comparisons made in the fourth phase will draw conclusions and, in turn, the recommendations that will emerge as conclusions.

1.5. Roadmap

In first chapter the introduction will be made, that is, a clear vision of what the work consists of. It will also be explained what the objectives of this work are, and the motivations that led to do the same, i.e. will be summarized the problems that exist today applying round-trip engineering in software development and how I will try to give recommendations to try to minimize the problems. choosing the most appropriate framework. Here also was defined the structure of the thesis and the methodology that will be applied.

Chapter two of this dissertation provide crucial overview of the concepts to supporting the UML round-trip engineering modeling tool. First, the background will be explained what it is UML involvement in software development. Next will be explained in detail what is model and metamodels and metamodeling as approach of Model Driven Development (MDD), followed by model transformation and evolution. of the software. It will also be summarized the works of other authors and explained how their work influenced to make decisions in my work, saying in other words the related work will be done.

The purpose of chapter three is to define benchmarking that will be used to make comparison of the round-trip engineering tools. After defining the benchmark model will be made a guide of the round-trip engineering tools setting up the development environment to carry out the experiments each tool will be studied from a quantitative and qualitative point of view.

Chapter four will have a qualitative and quantitative analysis, tool-type assessment framework consisting of a series of rigorous, metric assessments to cover forward and reverse-engineering per tool.

Finally, in chapter five we will answer the proposed research question and present the chosen UML modeling tools. It will also make a clear recommendation of the best round-trip engineering tool according to the assessment tools used.

Chapter 2 – Literature Review

2.1. Background

2.1.1 Concepts of Unified Modeling Language and Metamodels

The Unified Modeling Language is one of the most successful cases in the Information Technology (IT) industry, being used to outline the requirements of an IT system. Now, UML is used in a variety of ways by professionals with different backgrounds.

Since UML was standardized, the language has been extended into full System Modeling Language (SysML) to solve system development problems. Programmers, business analysts, architects, systems analysts and other stakeholders that rely of UML must learn the language, for them, UML diagrams are not simple images for communication and documentation purposes is a view of a formally defined mode (Booch, Rumbaugh, Jacobson, 2006). Software systems models are constructed, visualized, documented and specified using UML which is a system of language and notation. The UML adapts itself to technical systems and commercial systems and broadens a wide range of applications.

A methodology is considering a structure and how to specific a domain, in organizational environment and others. The UML can be used within a methodology and can form a basis for a distinct approach, since it is a defined set of modeling and construction of a uniform and semantic declaration.

Saying this we can consider the UML achieved two important aspects, first, it ended with the differences between previous modeling methods; second, it unified the perspectives between many systems of different types (business x software), development phases (requirements analysis, planning and implementation) and internal concepts.

The UML language is widely used in the so-called metamodel. The reason for the meta prefix is that the language resides at a level of abstraction above the UML model that the users uses. Now, the metamodel may seem confusing and illogical at first because the metamodel is the UML class model that describes the UML. In other words, the UML is defined in UML. Following the same line of reasoning, we have that a model is an instance of a metamodel. See a practical example in (Figure 2) (OMG, 2009).

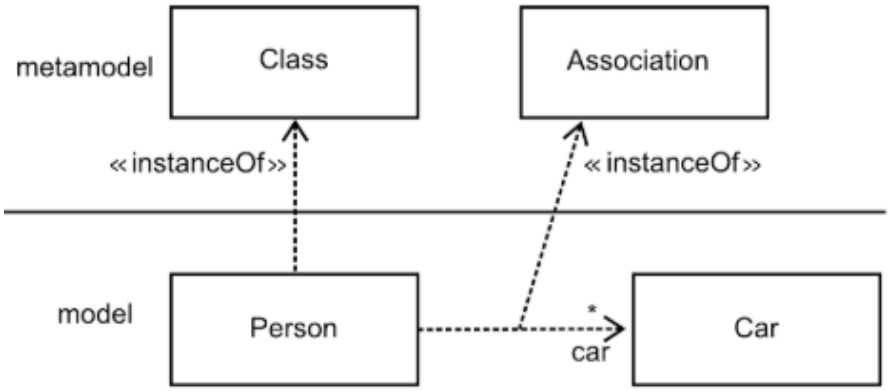

Figure 2 – Example a model is an instance of a metamodel (UML 2.2, 2015)

The class models provide quickly reached, when is defining a formal language like the UML. For this reason, the particularization of the UML describes formal limitations mainly in OCL, and the comments for each element allow to specify the semantics in more detail. New elements of the UML model are contained in the UML metamodel. In fact, only the subset of class modeling is required. In turn, this set is described in its own model, the meta-metamodel (OMG, 2007).

This means that we have a meta-metamodel and a metamodel. In a way, we also have the models of UML users, models what are used, which include other model, model of objects based on model. Thus, in total, it has four models, which will origin a four- layer UML architecture.

The model in the four-layer architecture of the UML, except the highest level, is an instance of one at the highest level. Firstly, the data user, refers to the data manipulated by the software (Alanen, 2005). User data models are called user concept models and are a level above the user's data level. Models of user concept models are models of UML concepts (Jouault and Bézivin, 2006). These are model of models and therefore called metamodels. A metamodel is a model of a modeling language. this is also a model whose elements are types in another model. The meta-metamodels are at the highest level of the modeling infrastructure, the Meta-Object Facility (MOF) (Alanen and Porres, 2008), (OMG, 2007).

To outgrow this issue, the OMG proposed a meta-model, the Meta Object Facility (MOF) which offers concepts for defining model language. It essentially provides a reduced subset of UML class diagrams, consisting of packages, classes, attributes, associations and operations. With these elements, complex modelling languages, such as

the UML itself can be described. Even the MOF can be defined itself in terms of these elements and thus forms a closure of an entire stack of half layers (Figure 3).

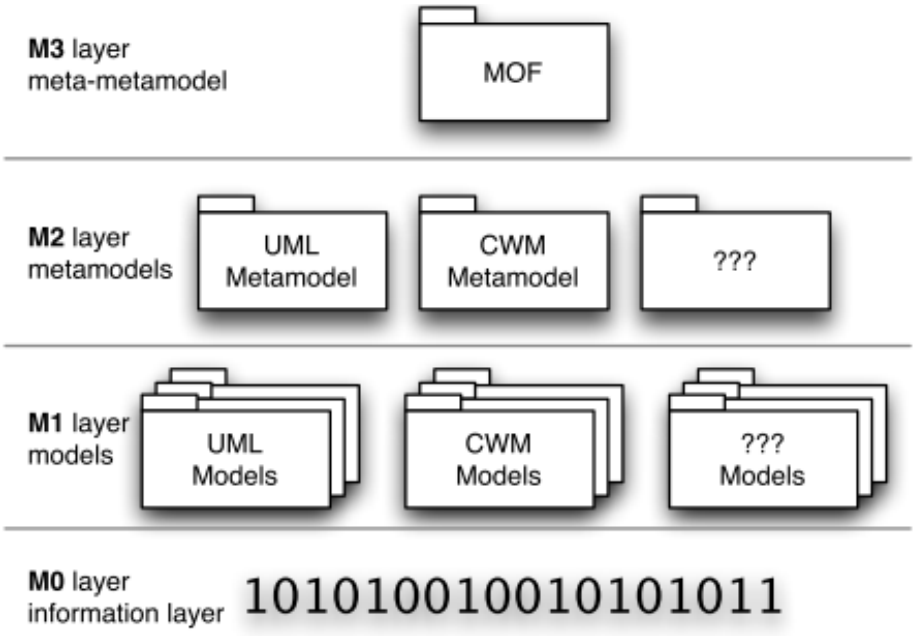

Figure 3 – Conceptual Layers in MOF (Hettel, 2010)

Models represent a system, they can be: Structural - diagram class and collaboration; Behavioral - state machine, activity diagram and sequence; Physical - component and deployment diagrams) of external functionalities and instances use case and object diagram (Terrasse, 2001). Models serve different purposes: Establish a clear understanding of the problem; Communicate a clear vision of the problem and solution; Generate low level implementations from high level models. (Brown, 2006). To make automation, models must have a defined meaning. A language consists of syntax and semantics. The syntax can be machine-centric or human. “Semantics define what the syntax means by linking the syntax to a semantic domain, rather like arithmetic expressions “mean” numbers “.

The models are developed in problem domains and solutions (Brown, 2005). The path from the problem domain to the solution domain requires an understanding of architecture. To get the logical architecture implementation in relation to the application in development, some levels of abstraction must be crossed (Cook, 2004). "Each of the views or models in the system captures the structure or behavior of the system with a specific abstraction depth" (Sendall and Kozaczynski, 2003).

According to Mellor, Clark and Futagami Model-Driven Development (MDD) (2003) “automates the transformation of models from one form to another, express each model, both source and target, in some language. Must be defined the two languages someway, because modeling is an appropriate formalism to formalize knowledge, we can define a modeling language’s syntax and semantics by building a model of the modeling language so-called metamodel”.

The models allow the efficiency and effectiveness of programming to be maintained over time, which means that the code change can be made with the least time consumption and the highest probability of success. In other situations, models are needed when establishing the requirements as well as obtaining certification in software development. Communication between developers and analysts can be greatly accelerated if models are used (Hailpern and Tarr, 2006). Templates can also function as a memory artifact when some developers are replaced by others who need to keep pace with what has been developed before. The same happens with solutions that must be developed over long periods of time. Following this reasoning, the Software Engineering approach MDD makes sense. MDD imposes structuring process of software development around models adequate to each one of the moments within that process.

2.1.2 MDD and MDA

MDD commonly is used to describe software development approaches in which abstract models of software systems are created and systematically transformed into implementations. As the modeling of the system evolves, transformations will be necessary to obtain models at different levels of abstraction (France and Rumpe 2007). "MDA distinguishes between independent models of its implementation platform and those that are not". MDE technologies tend to focus on the production of deployment and deployment artifacts from detailed design models (Hailpern and Tarr, 2006). In the modeling perspective, the systems implementation is done from models. Here, model generation can be done by applying standards. From the model-only perspective, models in general are just an artifact to understand the problem or solution.

As for language definition, Atkinson and Kühne (2003) divided the concept into four associated concepts: abstract and concrete syntax, good training, and semantics. Abstract syntax is equivalent to metamodels. The concrete syntax is equivalent to UML notation. Well-formed is equivalent to restrictions in abstract syntax Object Constraint Language

(OLC) for example: Semantics is the description of the meaning of a model in natural language (OMG, 2007).

MDE an incarnation of Model Driven Architecture (MDA) where the Object Management Group (OMG) promotes the idea that models, rather than code, are the main citizens in the software development process. The problem that needs to be solved by the application is "coded" in a domain model called the Independent Computing Model (CIM), where is described concepts such as clients and their relevant relationships in the application domain (OMG, 2005). This model is then refined into a Platform Independent Model (PIM), which introduces some parts of the solution, but is not yet associated to any computing platform. There is not restricted definition of what constitutes a platform that depends on the context.

Although the MDA view specifies only three distinct types of models and two refinement steps, in practice, several models and various modeling languages may be employed to better describe the system from various points of view and abstraction layers. For example, to design a Service Oriented Architecture (SOA) where is based application for a supply chain, one can begin designing the interactions between the different partners. From there, the local processes executed by the various partners can be derived. They can be enriched with additional activities and then translated into an execution format, such as Business Process Execution Language (BPEL).

2.1.3 Concepts to Understand Round-Trip Engineering

The purpose of round-trip engineering is to keep several artifacts up-to-date and consistent in propagating changes between artifacts. Making the artifacts consistent by propagating the changes is also called synchronization. Round-trip engineering is a special case of synchronization that can propagate changes in multiple directions, from models to codes, and vice versa. Round-trip engineering is difficult to achieve in a general scenario, due to the complexity of the mappings between the artifacts.

Perspectives on modeling are always around the code, without some code or some state between (Brown, 2005). From the point of view of the code, it is difficult to manage the evolution of the solutions due to problems of scale and complexity that they created. In the code perspective, the models need to understand the structure and behavior of the code. The model is another representation of the code. From a round-trip engineering

perspective, code generation is done from the model, and the model is updated whenever the code is changed.

The ability to trace new and changed requirements to their impacted components provides critical support for managing change in an evolving software system. Tracing is a standard abstraction stereotype primarily used to trace requirements and changes to models for elements or sets of elements that represent the same concept in different models. Thus, tracking is an "intermodal" relationship. These tracking dependencies between models are usually represented by dependencies between elements contained in models. The direction of the trace (i.e. customer and vendor designation) is at the discretion of the modeler, and since model changes can occur in both directions, the direction of dependency can often be ignored. Mapping specifies the relationship between the two but is rarely computable and is usually informal.

Shandall states that "round-trip engineering is a challenging task that will become an important facilitator for many approaches to Model-Driven Software Development." Round-trip engineering involves the synchronization and maintenance of the model, allowing software engineers to move freely between different representations. This round-trip engineering vision is only realized in some limited tools due to maintenance difficulties (Sendall and Küster, 2004).

We can now provide information about round-trip engineering and describe a situation that may emerge in round-trip engineering, but this does not occur in forward or reverse engineering. Several environment development tools support graphical models, which are usually subsets of the possible Unified Modeling Language models. Many of these tools provide automated support for generating code from UML class diagrams and generating UML class diagrams from code. The first task is called advanced engineering and the second is reverse engineering.

Forward Engineering involves the generation of one or more software artifacts closer to the detail level of the final system compared to the input artifacts, and Reverse Engineering involves the generation of one or more software artifacts that abstract certain details and possibly present themselves in a in a different way, the input artifacts to retrieve any information lost in the routing step. Some tools also support round-trip engineering, which involves several stages of forward and reverse engineering, so that software artifacts, such as programming language code and UML class diagrams, are synchronized whenever changes occur (OMG, 2003).

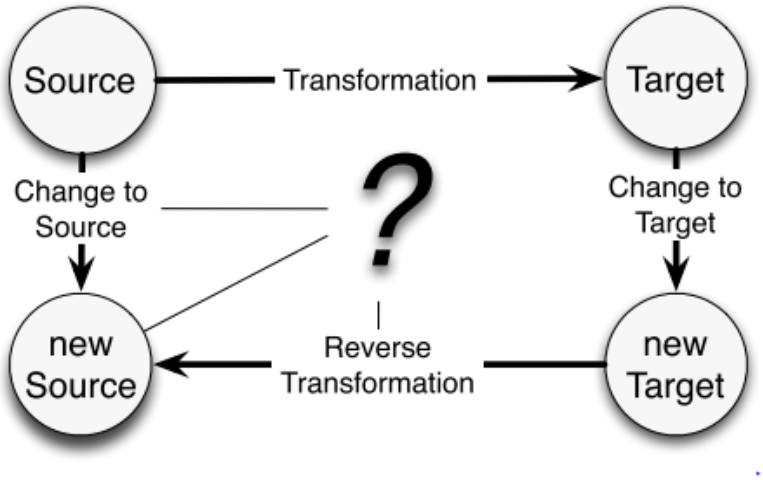

Figure 4 – Model Round-trip Engineering (Hettel, 2010)

Many existing development tools offer very limited support for round-trip engineering. This is probably a consequence of the difficulty in keeping several artifacts up to date. It may be noted that it may not be necessary to allow multiple artifacts to be changed during the same time, this will certainly simplify the problem if only one artifact can be changed and the other is simply viewed or sojourn for reading. For example, round-trip engineering is not normally required between the programming language code and the binary code, because it is assumed that the binary code will not be changed.

This task is typically only isolated transformation, in which any information in the target artifact is not considered and usually a new artifact is created, which is likely to replace the previous version if it exists. In some cases, forward and reverse engineering can be optimized so that only incremental transformation is performed or whenever updates exist, that is, only changed modules are transformed, rather than all artifacts, for example, incremental compilation.

From another point of view, round-trip engineering requires the information in the destination artifact is preserved and not modified on the return trip. This also indicates that the attempt behind round-trip engineering is to reconcile the models rather than simply turn them into a certain direction. For example, in round-trip class diagrams and Java UML class diagrams, it would often be undesirable for form and join names to be changed in the class diagram if an unrelated change was made to Java code. In this case, the Java programming language is a class-based programming language. This type of

situation arises whenever there is enough information in the source artifacts to reconstruct the target artifacts in their former form (Kleppe, 2003).

Given a set of models, round-trip engineering is a technique that allows the software to move free between these models and modify them as appropriate, where changes made in each are reflected in the others, according to the model. Round-trip engineering can be divided into three steps: deciding whether a model under consideration has been modified; you decide whether the changes cause some inconsistency with the other models; when inconsistencies are detected, update the other models so that they become consistent again.

Given a set of models, round trip engineering is a technique that allows the software developer to move freely between models and change them as needed, where changes made in each are reflected in the others, keeping each consistent. In order to decide whether the model is consistent or not, you must first understand what makes the models consistent with each other. This requires a semantic knowledge of each model and a definition of the relationships between them. But the big problem is that the semantic knowledge of each model and the definition of the relationships between them is usually shared informally meaning that there will be problems in the formal execution of a task and the lack of tool support. Once inconsistencies are detected, changes must be made to make the models consistent again. This problem has been well studied in the field of software engineering and is often referred to as inconsistency management (Spanoudakis and Zisman, 2001). The main challenge is to make them consistent and ensure that all possible situations that may arise have been covered and that the effect of updating models in each case leads to stability and consistency.

2.1.1 Description of Class Diagram

An important part of UML is its semantics document (France, Evans, Lano and Rumpe, 1998), which attempts to give a sound semantic basis to its diagrams. Meta-models are used to describe the syntax of its static and behavioral models, while semantic details are expressed in informal English. Unfortunately, the informal nature of these semantics is inadequate for justifying formal analysis techniques for UML. Thus, we need to develop a more precise semantic description. UML is also a large modelling language.

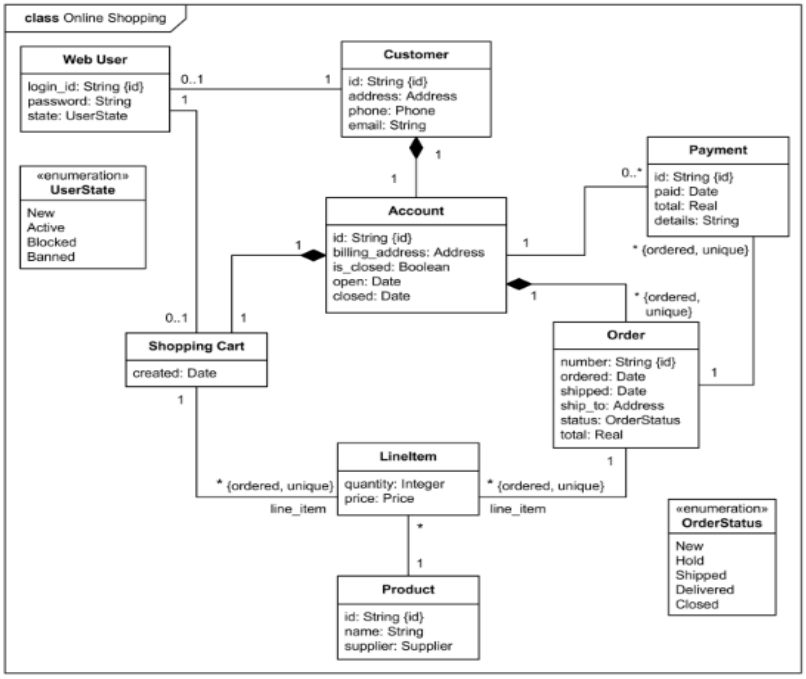

Figure 5 – Example of UML class diagram (UML Example, 2013)

The static UML model is visually represented by a class diagram. Its purpose is to graphically depict the relationships holding among objects manipulated by a system. In software, many programming languages directly support the concept of a class. That means the abstractions you create can often be mapped directly to a programming language, even if these are abstractions of non-software things. The UML provides a graphical representation of class, as well. This notation permits you to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations (Folwer,1997).

Every class must have a name that distinguishes it from other classes. A name is a textual string. That name alone is known as a path name is the class name prefixed by the name of the package. Class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a class name.

An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. At a given moment, an object of a class will have specific values for every one of its class's attributes. Graphically, attributes are listed in a

compartment just below the class name. Attributes may be drawn showing only their names.

An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all. Operations may be drawn showing only their names. An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in move or isEmpty (France, Bruel, and Larrondo-Petrie, 2000).

In Unified Modelling Language (UML), interfaces are model elements that define sets of operations that other model elements, such as classes, or components must implement. An implementing model element realizes an interface by overriding each of the operations that the interface declares.

Interfaces support the hiding of information and protect client code by publicly declaring certain behavior or services. Classes or components that realize the interfaces by implementing this behavior simplify the development of applications because developers who write client code need to know only about the interfaces, not about the details of the implementation. If you replace classes, or components that implement interfaces, in your model, you do not need to redesign your application if the new model elements implement the same interfaces.

In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships. In object-oriented modeling, there are three kinds of relationships that are most important: dependencies, generalizations, and associations (Booch, Rumbaugh and Jacobson 2000).

A dependency is a using relationship that states that a change in specification of one thing (for example, class Event) may affect another thing that uses it (for example, class Window), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another. Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature

of an operation. In the UML you can also create dependencies among many other things, especially notes and packages.

A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often but not always the child has attributes and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent. Use generalizations when you want to show parent/child relationships.

An association is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations. Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. You can show a multiplicity of exactly one (1), zero or one (0...1), many (0...*), or one or more (1...*) or more the one in exact number.

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes, you will want to model a relationship, in which one class represents a larger thing, which consists of smaller things. This kind of relationship is called aggregation, which represents a relationship, meaning that an object of the whole has objects of the part. Aggregation is just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end.

2.1.2 Description of Statechart Diagram

Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. A statechart diagram shows a state machine. An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state. Thus, statechart diagrams are useful in modeling the lifetime of an object and shows flow of control from state to state. Statechart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, construct, and document the dynamics of an individual object. Statechart diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering (Booch, Rumbaugh, Jacobson, 2000).

A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events. A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs, and specified conditions are satisfied.

A statechart diagram is basically a projection of the elements found in a state machine. This means that statechart diagrams may contain branches, forks, joins, action states, activity states, objects, initial states, final states, history states, and so on. The State-Machines package defines a set of concepts in order to model discrete behavior of the system through a finite state transition mechanism (Sunitha and Samuel, 2016).

Each state models a period during the life of an object during which it satisfies certain conditions, performs some action, or waits for some event. A state becomes active when it is entered as a result of some transition and becomes inactive if it is exited as a result of a transition. A transition is a directed relationship between a source state and a target state indicating that an instance in the source state will enter the target state and performs certain actions when a specified event occurs provided that certain specified conditions are satisfied. The trigger for a transition is the occurrence of the event labeling the

transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. When an event occurs, it may cause the action of transition that takes the object to a new state. Events are processed one at a time. If an event does not trigger any transition it is discarded (Sekerinsk, 2009).

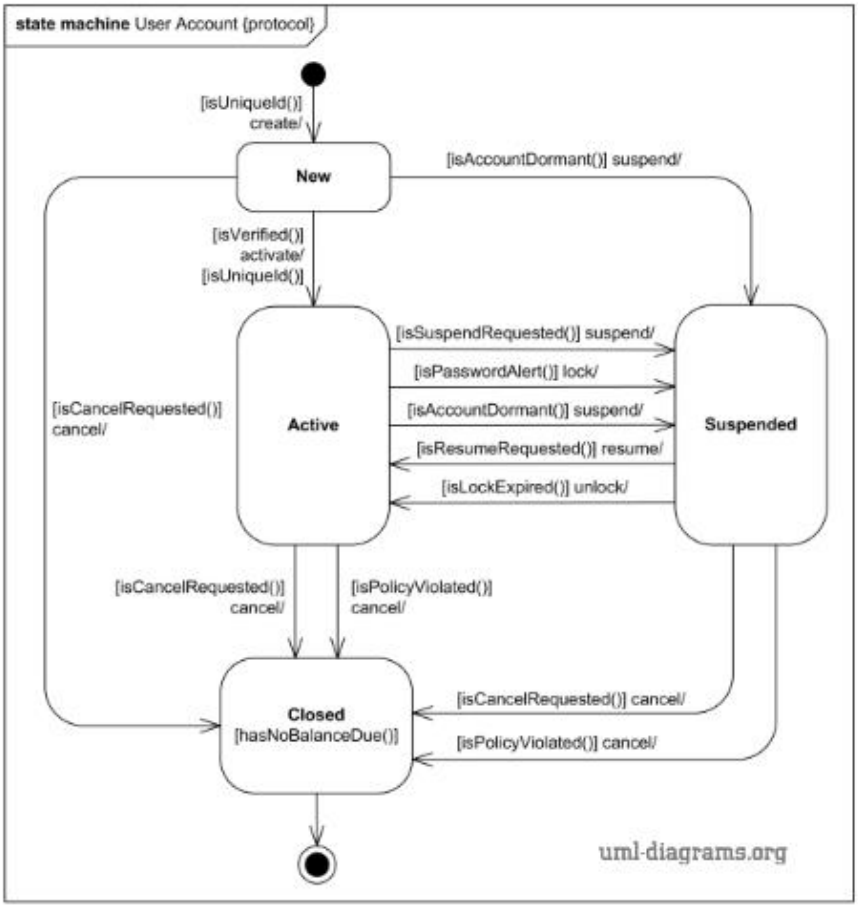

Figure 6 – Example of UML state-machine diagram (UML Example, 2013)

A state is an abstract meta-class that models a situation during which some invariant condition hold. This invariant may represent a static situation such as an object waiting for some external event occur. However, it can also model dynamic conditions such as the process of performing some activity, that is the model element under consideration enters the state when activity commences and leaves it as soon as the activity is completed. A composite state is a state that contains other states vertices. The association between the composite and the contained vertices is a composition. Hence, a state vertex can be a part of at most one composite state. A simple state is state that does not have sub states.

An event is the specification of type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration. Strictly speaking, the term event is used to refer to type and not to an instance of the type. However, on occasion, where the meaning is clear from the context, the term is also used to refer to an event instance. An event can have the association parameter, that specifies this list of parameters defined for the event. A guard is a Boolean expression that is attached to transition as a fine-grained control over its firing. The guard is evaluated when an event instance is dispatched by the state machine. If the guard is true at the time, the transition is enabled, otherwise is disabled (Pham, Radermacher, Gérard, and Li, 2017).

2.1.3 Description of Component Diagram

Over time and across successive releases of UML, the original UML meaning of components was mostly lost. UML 2 officially changes the essential meaning of the component concept; in UML 2, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces.

Although the UML 2 specification does not strictly state it, components are larger design units that represent things that will typically be implemented using replaceable modules. The idea is that you can easily reuse and/or substitute a different component implementation in your designs because a component encapsulates behavior and implements specified interfaces. In UML 2 the physical items are now called artifacts. An artifact is a physical unit, such as a file, executable, script, database, etc.

A component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems. A component diagram shows the organization and dependencies among a set of components. You use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents. Component diagrams are essentially class diagrams that focus on a system's components. Component diagrams are not only important for visualizing, specifying, and documenting component-based systems, but also for constructing executable systems through forward and reverse engineering (Donald, 2004).

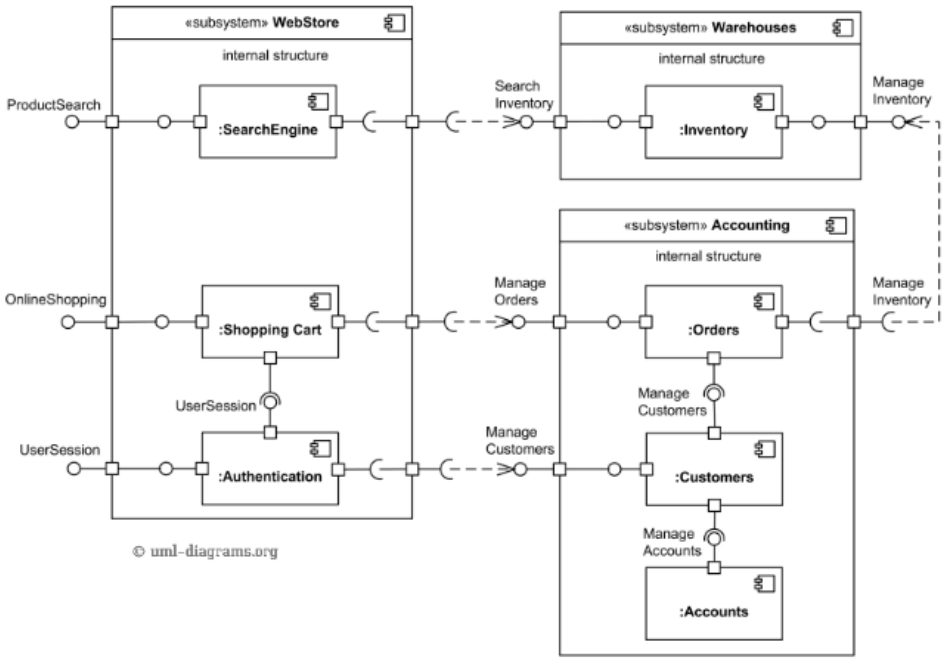

Figure 7 – Example of UML component diagram (UML Example, 2013)

Component diagrams commonly contain Components, Interfaces, Dependency, generalization, association, realization relationships and may contain notes and constraints like other diagrams. Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you will want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems (Booch, Rumbaugh and Jacobson, 2000).

When modeling large software systems, it is common to break the software into manageable subsystems. UML provides the component classifier for exactly this purpose. A component is a replaceable, executable piece of a larger system whose implementation details are hidden. The functionality provided by a component is specified by a set of provided interfaces that the component realizes. In addition to providing interfaces, a component may require interfaces in order to function. These are called required interfaces.

A component that represents a business concept. An entity component typically passes information in and out of interfaces and is often persisted. Entities don't typically have any functionality, or service capabilities, associated with them; they are usually just for data storage and retrieval. A component that can fulfill functional requests (as opposed to an entity component). A process component is transaction-based and typically has

some type of state associated with it (as opposed to stateless service components). A stateless component that can fulfill functional requests. Service components are requests. Service components are rarely persisted because they contain no state.

Forward engineering and reverse engineering components are direct, because components are themselves physical things (executables, libraries, tables, files, and documents) that are therefore close to the running system. When you forward engineer a class or a collaboration, you forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration. Similarly, when you reverse engineer source code, binary libraries, or executables, you reverse engineer to a component or set of components that, in turn, trace to classes or collaborations. Choosing to forward engineer (the creation of code from a model) a class or collaboration to source code, a binary library, or an executable is a mapping decision you must make. You will want to take your logical models to source code if you are interested in controlling the configuration management of files that are then manipulated by a development environment. You willl want to take your logical models directly to binary libraries or executables if you are interested in managing the components that you will deploy on a running system. In some cases, you will want to do both. A class or collaboration may be denoted by source code, as well as by a binary library or executable.

Reverse engineering a component diagram is not a perfect process because there is always a loss of information. From source code, you can reverse engineer back to classes; this is the most common thing you will do. Reverse engineering source code to components will uncover compilation dependencies among those files. For binary libraries, the best you can hope for is to denote the library as a component and then discover its interfaces by reverse engineering. This is the second most common thing you will do with component diagrams. In fact, this is a useful way to approach a set of new libraries that may be otherwise poorly documented. For executables, the best you can hope for is to denote the executable as a component and then disassemble its code something you will rarely need to do unless you work in assembly language.

2.1.4 Description of Activity Diagram

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity. You use activity diagrams to model the dynamic aspects of a

system. For the most part, this involves modeling the sequential or concurrent steps in a computational process. With an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control. Activity diagrams may stand alone to visualize, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation.

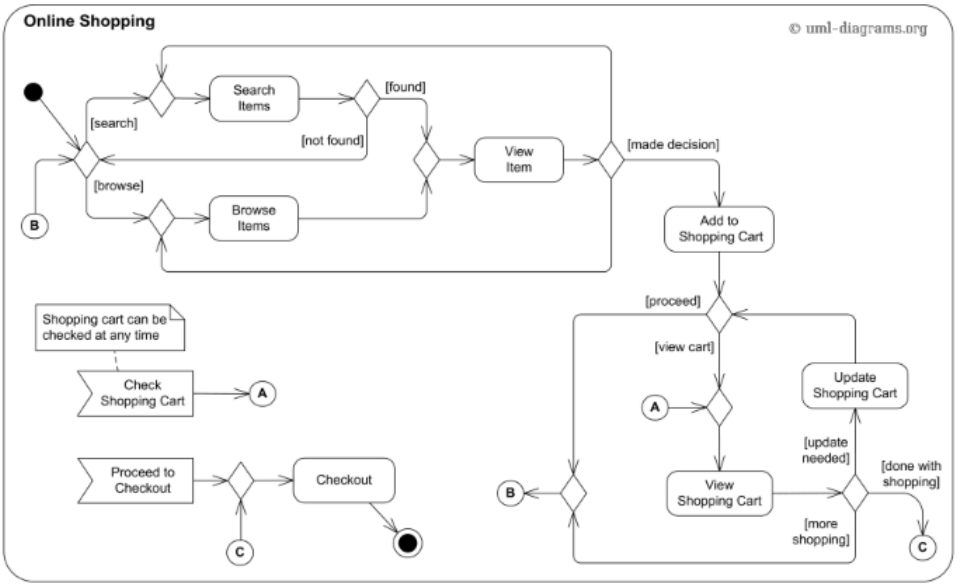

Figure 8 – Example of UML activity diagram (UML Example, 2013)

Whereas interaction diagrams highlight the flow of control from object to object, activity diagrams highlight the flow of control from activity to activity. An activity is an execution within a state machine. Activities ultimately result in some action, which is made up of executable computations that results in a change in state of the system or the return of a value. Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Although that's a tremendous simplification of what really goes on in a construction process, it does capture the critical path of the workflow. In a real project, there are lots of parallel activities among various trades. You will also encounter conditions and branches and might even be loops.

How do you best model a workflow or an operation, which are aspects of the system's dynamics? The answer is that you have two basic choices, like the use of Gantt and Pert charts. On the one hand, you can build up storyboards of scenarios, involving the interaction of certain interesting objects and the messages that may be dispatched

among them. (Booch and Rumbaugh, 2004). In the UML, you can model these storyboards in two ways: by highlight the time ordering of messages (using sequence diagrams) or by emphasizing the structural relationships among the objects that interact (using collaboration diagrams). Interaction diagrams such as these are akin to Gantt charts, which focus on the objects that carry out some activity over time.

On the other hand, you can model these dynamic aspects using activity diagrams, which focus first on the activities that take place among objects. In that regard, activity diagrams are akin to Pert charts. An activity diagram is essentially a flowchart that highlight the activity that takes place over time. You can think of an activity diagram as an interaction diagram turned inside out. An interaction diagram looks at the objects that pass messages; an activity diagram looks at the operations that are passed among objects. The semantic difference is subtle, but it results in a very different way of looking at the world.

An activity diagram shows the flow from activity to activity. Activities result in some action, which is made up of executable computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression. (Booch and Rumbaugh, 200).

In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object. These executables, computations are called action states because they are states of the system, each representing the execution of an action. Inside that shape, you may write any expression.

Action states and activity states are just special kinds of states in a state machine. When you enter an action or activity state, you simply perform the action or the activity; when you finish, control passes to the next action or activity. Activity states are somewhat of a shorthand, therefore. Nonetheless, activity states are important because they help you break complex computations into parts, in the same manner as you use operations to group and reuse expressions.

Semantically, these are called trigger, or completion, transitions because control passes immediately once the work of the source state is done. Once the action of a given

source state completes, you execute that state's exit action. Next, and without delay, control follows the transition and passes on to the next action or activity state. You execute that state's entry action, then you perform the action or activity of the target state, again following the next transition once that state's work is done. This flow of control continues indefinitely until you encounter a stop state.

The definition of plans and activities are similar. Plans are composed of actions and define the order in which they can be executed, thus activity diagrams can be used to model plans. Like an activity, a plan can be illustrated by using three different representations. The actions and edges that compose and describe the plan are modeled inside a round-corned rectangle identified by the name of the plan.

In UML activity diagrams, it is possible to define an action in two different ways. An action can be identified only by its name or it can be described using an application description language. When the designer defines an action, he is specifying a component that will implement a given functionality. Besides, a plan is just a logical sequence of actions. The implementation of these actions can be independent, to maximize loose coupling and action reuse.

To illustrate this idea, actions could be described and implemented using a services approach. In such approach, all actions are defined as services, which are black boxes. In a more general sense, the action interface is invokable. This means that it is irrelevant if an action is local or remote, what interconnect scheme or protocol is used to affect the invocation, or what infrastructure components are required to make the connection. In this approach, an action could be further specified using Web Services Description Language (WSDL) (Lind, 2002). It is possible to notice how vast the description of actions can become by using WSDL. To solve such problem, we propose to identify actions by describing their names and identifying the URL where their WSDL descriptions are available.

2.1.5 Description of Sequence Diagram

Interaction diagrams are defined by UML to emphasize the communication between objects, not the data manipulation associated with that communication. Interaction diagrams focus on specific messages between objects and how these messages come together to realize functionality. While composite structures show what objects fit together to fulfill a requirement, interaction diagrams show exactly how those objects will

realize it. Sequence diagrams are a type of interaction diagram that highlight the type and order of messages passed between elements during execution. Sequence diagrams are the most common type of interaction diagram and are very intuitive to new users of UML.

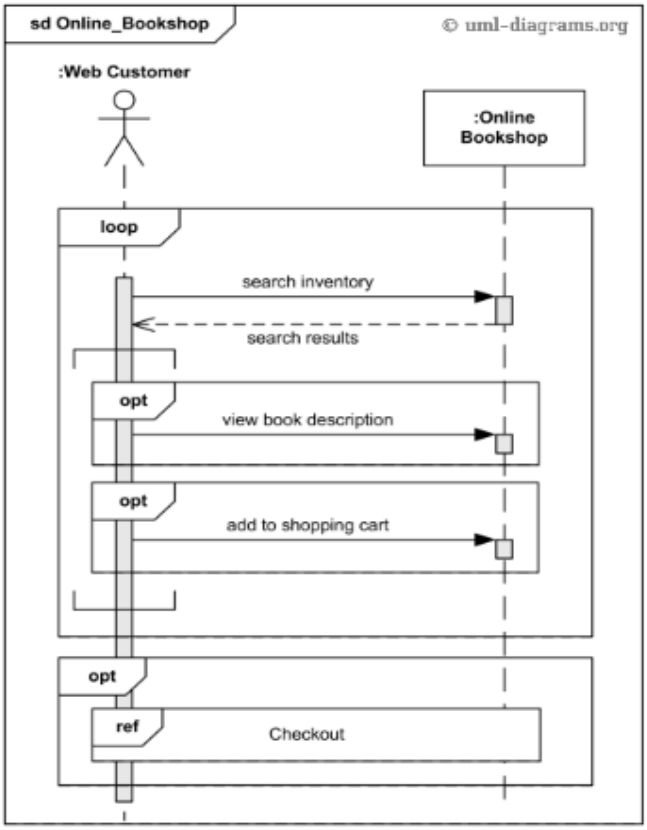

Figure 9 – Example of UML sequence diagram (UML Example, 2013)

UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems (Rumbaugh, Jacobson and Booch, 1999). It is more common to use statecharts and sequence diagrams to specify the behaviors of the internal structural pieces of the system. Nevertheless, they are useful for capturing, specifying, and illustrating requirements on the system as well. As is very common, the sequence diagram describes the use case, it is preconditions and postconditions, and keeps a running dialog explaining what is going on as the scenario unfolds.

Statecharts work very well for defining the behavior of reactive objects. The UML provide sequence diagrams to show how object collaborate by sending message (calling operations) and events to each other. The vertical lines represent objects (not classes) during the execution of the system. The arrowed lines are messages (either calls to operations on objects or events sent to objects). The hatched area on the left is called a "collaboration boundary" and represents all objects in the universe other than those

explicitly represented as instance lines on the diagram. Time flows down the page. It is easy to see how these objects work together to produce the desired system effect. (Firley, Huhn, Diethers, Gehrke and Goltz, 2000) The goals what system achieves does not appear on the diagram, because that is the job for implementation and design, not for requirements capture.

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. Much like the class diagram, developers typically think sequence diagrams were meant exclusively for them. However, an organization's business staff can find sequence diagrams useful to communicate how the business currently works by showing how various business objects interact. Besides documenting an organization's current affairs, a business-level sequence diagram can be used as a requirements document to communicate requirements for a future system implementation. During the requirements phase of a project, analysts can take use cases to the next level by providing a more formal level of refinement. When that occurs, use cases are often refined into one or more sequence diagrams.

The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on messages themselves and more on the order in which messages occur; nevertheless, most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

When drawing a sequence diagram, lifeline notation elements are placed across the top of the diagram. Lifelines represent either roles or object instances that participate in the sequence being modeled. Lifelines are drawn as a box with a dashed line descending from the center of the bottom edge. The lifeline's name is placed inside the box.

The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability. Subsequent messages are then added to the diagram slightly lower than the previous message. To show an object sending a message to another object, you draw a line to the receiving object with a solid arrowhead or with a stick arrowhead. The message/method name is placed above the arrowed line.

The message that is being sent to the receiving object represents an operation/method that the receiving object's class implements.

Occasionally you will need to model a repetitive sequence. In UML 2, modeling a repeating sequence has been improved with the addition of the loop combination fragment. The loop combination fragment is very similar in appearance to the option combination fragment. You draw a frame, and in the frame's name box the text "loop" is placed. Inside the frame's content area, the loop's guard is placed towards the top left corner, on top of a lifeline. The sequence diagram content area. In a loop, a guard can have two special conditions tested against in addition to the standard Boolean test. The special guard conditions are minimum iterations written as "min int = [the number]" (and maximum iterations written as "max int = [the number]". With a minimum iterations guard, the loop must execute at least the number of times indicated, whereas with a maximum iteration guard the number of loop executions cannot exceed the number.

The sequence diagram is a good diagram to use to document a system's requirements and to flush out a system's design. The reason the sequence diagram is so useful is because it shows the interaction logic between the objects in the system in the time order that the interactions take place.

2.1.6 Description of Use case Diagram

During requirements analysis, Use Case diagrams help to identify the actors and to define by means of Use Cases the behavior of a system (Jacobson, 1992). Use Case modeling is accepted and widely used in industry. If Use Case diagrams support the modeling of functional variability, they can also be used to describe common and variable behavioral characteristics of a Software Product Line. Hence, we take a brief look on the UML Use Case modeling elements. Beside actors and Use Cases UML defines a small set of relationships to structure actors and Use Cases. Use Cases may be related to other Use Cases by the following relationships:

• Extend An extend relationship implies that a Use Case may extend the behavior described in another Use Case, ruled by a condition.

• Include An include relationship means that a Use Case includes the behavior described in another Use Case.

• Generalization between Use Cases means that the child is a more specific form of the parent Use Case. The child inherits all features and associations of the parent and may add new features and associations (OMG, 2001).

A Use Case describes the interaction between a system and its environment. A Use Case defines a goal-oriented set of interactions between external actors and the system under consideration. The term actor is used to describe the person or system that has a goal against the system under discussion. A primary actor triggers the system behavior in order to achieve a certain goal. A secondary actor interacts with the system but does not trigger the Use Case. A Use Case is completed successfully when that goal is satisfied. Use Case descriptions also include possible extensions to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure in completing the service in case of exceptional behavior, error handling, etc. A complete set of Use Cases specifies all the different ways to use the system, and therefore defines the whole required behavior of the system. Generally, Use Case steps are written in an easy-to understand, structured narrative using the vocabulary of the domain. A scenario is an instance of a Use Case and represents a single path through the Use Case. Thus, there exists a scenario for the main flow through the Use Case, and as many other scenarios as the possible variations of flow through the Use Case. Scenarios may also be depicted in a graphical form using UML Sequence Diagrams.

Though it is possible to introduce stereotypes to mark relationships between Use Cases, stereotypes typically lack of a defined semantic and may have different meanings in different contexts. Furthermore, it is possible to define constraints between Use Cases, but their semantics may vary in different contexts, too. Therefore, new Use Case modeling elements are needed to explicitly express all types of variability described above. It should be mentioned, that feature graphs, which represent characteristics of the system, can be used to complement Use Case modeling and to organize the results of the commonality and variability analysis in preparation for reuse (Clements and Northtop, 2001). These two notations allow to model variability in a domain from two different perspectives. Variability exists in features and in sequences of activities that means between product line members and at runtime of a specific product. Feature graphs are structure oriented because they describe the characteristics of a domain and the relationships between them. Furthermore, they are intended to show hierarchies in structures and visualizes dependencies between characteristics, though in a restricted way

because only dependencies between parent and child nodes can be expressed directly. Use Cases instead are suitable to model sequences of activities and they model therefore dynamic characteristics and dependencies between these activities. A suitable process in capturing variability in an unknown domain is to start with capturing Use Cases to find essential activities that are executed in that domain.

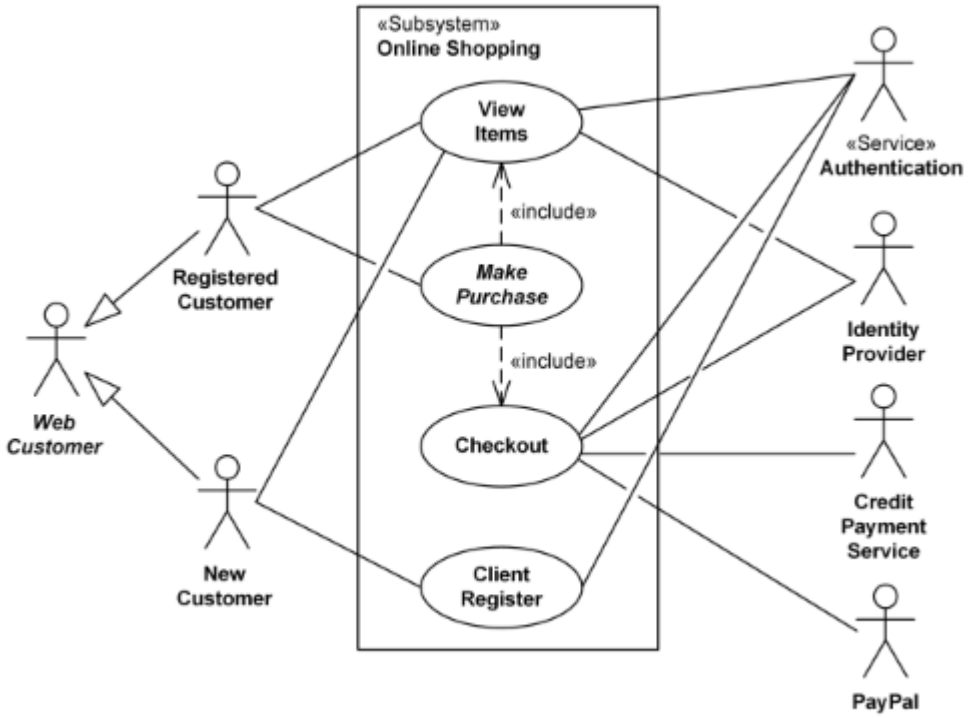

Figure 10 – Example of UML use case diagram (UML Example, 2013)

There is no standardized form for the content of a use case itself, the standard describes the graphical representation and the semantics of use case diagrams only. Use cases are fundamentally a text form although they can be written using flow charts or sequence charts (Cockburn, 2001). Use cases serve as a means of communication from one person to another, often among persons with no training in UML or software development. So, writing use cases in simple text is usually a good choice. There is no general agreement on the attributes use cases should have and on the level of description of the use cases. The use case is described with its actors, the triggers, which means the actors that can activate the use cases. The input and output of the use case are described and the post conditions and a success guarantee and a minimal guarantee are given. The main part of the use case is the main success scenario which describes what the use case does.

2.2. Related Work

Khaled compiled in (2009) some modeling tools in UML, defining the characteristics considered as the most important of each of them and comparing these tools based on these characteristics, which resulted in a recommendation of different tools for different purposes. However, Khaled's work did not have a high degree of completeness on the characteristics related to the roundtrip engineering approach, namely traceability, and consequent synchronization, between models and code, in both directions (model-code and code-model), in addition to that the set of evaluated tools was frankly reduced.

Khaled affirms: today UML is considered as a *de facto* standard in software development and is used in many domains ranging from scientific modeling to business modeling for such the architectures of the applications that are currently used are quite complex and need to use recent UML versions to be able to give support, since it has a better flexibility and support a greater number of models. Considering the needs of companies nowadays is to deliver a software in time and with quality, is necessary to use tools with round-trip engineering approach that in turn of the implementation of code which consequently facilitates and reduces the time of the development of the software.

The author Wang in his research work on about a flexible “UML Class Diagramming Tool for Eclipse, have used the Green tool which is a plug-in of the UML class diagrams editor for Eclipse, initially designed for modeling and design. The ease of use and the features that it has enabled it to become a robust tool. Green's live round-trip engineering feature allows users to generate code from UML class diagrams and *vice-versa*, and both are up to date when any changes are made.

But during the tests was found inconsistencies, the tool does not handle static internal classes with total precision which in turn can generate incompatible code. There have also been usability issues as the diagram can be created, covering multiple widths and screen heights, but there is no way to visualize the entire diagram. Refactoring an element represented in a diagram while the diagram is closed will cause the diagram to lose the element. Given that Wang covered multiple factors in their test like a quality of code and usability, we can consider the same factors for future experiments but with other tools.

Round-trip engineering integrates technology from various areas of software engineering. As already mentioned, inconsistency management is a technique for making models consistent again. The management of inconsistencies has been studied for

approaches from the point of view within software engineering, giving rise to several approaches. Approaches that study consistency management for UML models are the basis for a solid definition of round-trip engineering. We have already pointed out that the round-trip engineering base is a clear definition of the required consistency between the models, which is generally a goal of consistency management. In relation to UML models, this task is complicated by the absence of a formal semantics and a common development process.

After analyzing the conclusions and recommendations presented by Khaled, we can deduce that they cannot be used at present because the versions of frameworks used are old and in turn cannot support complex and modern architectures. But this author's research work is very important for future research work because it has left a basis that can be used. The choice of the diagrams of the study to be carried out was based on the work of Khlaed, because when she did the case study some of the tools used did not support UML 2.0, but 1.4, as there was an evolution between these versions, would be It is interesting to see if the conclusions drawn today will be similar to those of Khaled.

Considering, Wang covered several factors in his testing, such as code quality and usability, we can consider the same factors for future experiments. Detailing Wang's analysis we found that Wang did a quantitative study of the code, as well as a semantic study to see if it is consistent with the diagram. The same was applied when the diagram was updated, i.e. it was checked with the original diagram if it is and consists. So, to speak, the analysis work that Wang has done in his study, namely the quality of the code, verifying that it is consistent with the diagram, is a fundamental basis for the work that will be done.

Chapter 3 – Implementation of Methodology

3.1. Chapter introduction

MDA is undoubtedly a good paradigm to support teams during the software design and development process. This approach can be even more effective if tool support is close to the practices and process concepts used in application domains. Its core strength, in addition to strong UML 2 compliance, depends on its ability to harness the power of profile management, but also to customize tools for domain-specific applications and generate diagram source code. The case studies that will be explained below are used for experimental validation of the Papyrus, Modelio and Visual Paradigm tools described above.

3.2. Selection Criteria

The tools that were analyzed before deciding were Papyrus, Modelio, BoUML, MEtaEdit ++, Visual Paradigm Accelo (List UML tools, 2013), all support the chosen UML diagrams. They also support MDA which is responsible for the round-trip engineering method. Below will be explained the reason for choosing each tool.

The Papyrus tool was chosen because it is a UML2 graphical modeler that currently supports eight of the all diagrams. Because modeling is not enough to claim to be an MDA tool, it also provides code generation (C ++, Java) and facilitates the connection of external tools (planning analysis) to enable models to be the main artifacts. of the development process. There are studies where Papyrus has been used in similar experiments, so it was a point of reference and one of the reasons for choosing. Another reason is that it is an open source project based on an Eclipse environment, which is the most widely used tool for software development and offers the freedom to enhance and evolve the tool without relying on any extrenal framework. Papyrus is used in industry and research, making it a solid and reliable tool.

Visual Paradigm is a tool like Papyrus, but with specific characteristics. It is a professional tool and unlike Papyrus is a proprietary software which make the tool more stable and support guaranteed. It is used in industries for process automation. It has an integrated graphical editor for UML 2.x and can generate source code through MDA. Another reason is that it has direct integration with Eclipse, as it makes it easy to have everything centered on a single IDE. The Generic Modeling Environment (GME) is a configurable Meta-CASE tool providing toolkits for creating a Domain Specific modeling

environment, configuration is done by specifying the modeling paradigm metamodel that represents the modeling language of the application domain.

Modeling like Papyrus is an open source tool with its own IDE. And one reason for choosing was beyond a predefined set of model-driven code generation modules, Modelio provides a strong integrated MDA capability through its UML profile editor and rich Java API for metamodel access, model transformation and tool customization. Another reason is that modeling provides complete coverage to support modeling activities for each stakeholder within a company that in turn is reflected in the diagrams the tool supports. Its modeling support includes UML2, BPMN, Enterprise Architecture, SOA Architectures, Requirements Analysis, Dictionary and Business Rule Analysis, which in turn with the other tools, is used in industry and makes it a reliable tool.

3.3. Description of Selected Frameworks

3.3.1. Papyrus

As part of its Model-Driven Architecture (MDA) initiative, the Object Management Group (OMG) has provided a comprehensive set of standardized technology recommendations to support model-based development of software and systems in general. They cover key features such as meta-modeling, model transformations, and domain-specific and general-purpose modeling languages.

Therefore, the Eclipse platform, along with its Model Development Tools (MDT) subproject, is the environment of choice for developing open source modeling tools. To achieve this goal, industry-standard meta-models and tools for developing models based on these meta-models were implemented. The implementation of the UML2 metamodel was first. This component became the default implementation of the UML2 metamodel.

Papyrus is a tool that consists of several editors, primarily graphic editors, but is also completed with other editors such as text-based and model-based editors. All these editors allow simultaneous viewing of multiple diagrams of a given UML model. Modification of an element in one of the diagrams is immediately reflected in other diagrams showing that element. Papyrus is integrated into Eclipse as a single editor linked to a UML 2 model. Papyrus provides a main view, showing model diagrams and additional views, including an outline view, a property, and an aerial view.

The various diagrams are managed by Papyrus and not by Eclipse. Model diagrams can be arranged in tabbed views, and various tabbed views can be arranged side by side.

Such views can be created, resized, moved and deleted by dragging them to the desired position. Papyrus is highly customizable and allows you to add new diagram types developed using any Eclipse compatible technology (GEF, GMF, EMF and others). This is achieved through a plug-in mechanism of all diagrams (Eclipse Papyrus Project, 2015).

When designing a UML2 profile, you may need to customize one or more existing UML2 diagram editors. For this purpose, Papyrus supports customization of existing publishers, with the added ability to extend these customizations by adding relevant new tools to the stereotypes defined in the UML profile. For example, the SysML Requirements Diagram Editor is designed as a customization of the UML2 class diagram editor with additional features for direct manipulation of all concepts defined in the SysML Requirements Diagram. Finally, by embedding a profile in an Eclipse plug-in, a designer can also provide a specific property view that will simplify manipulation of stereotypes and their related properties. The outline editor and tool menu can also be customized to address domain-specific concerns appropriate to the profile (F. Bordeleau).

It was developed by the Model-driven Engineering Laboratory for Embedded Systems (LISE), which is part of France's Alternative Energy and Atomic Energy (CEA List). It supports software design by providing JAVA or C ++ code generation from models, including real-time systems. This happens on two different levels of abstraction. Support for component-based models. In this case, generation starts from a model that includes the definition of software components, hardware nodes, and deployment information. The latter consists of a definition of the components and nodes and an allocation between them. Code generation is done by a sequence of transformation steps. The model takes care of some specific aspects of the properties based on. Therefore, Papyrus currently supports eight of the diagrams described in the specification: Class Diagram, Diagram Components, Activity Diagram, State Machine Diagram, Use Case Diagram, Sequence Diagram.

3.3.2. Modelio

Modelio is primarily a modeling environment, supporting a wide variety of models and diagrams and providing model assistance and consistency checking capabilities. Modelio combines BPMN support and UML support into one tool, with dedicated diagrams to support business process modeling. The Java Designer module uses an RCP / Eclipse style project file structure and supports Java code generation and rollback,

Javadoc generation, and Java automation. Modelio can be extended to any modeling language, methodology, or technique just by adding modules to your configuration. You can use existing modules or develop your own.

Modelio is developed and maintained by Modeliosoft. Headquartered in Paris, France, Modeliosoft provides training and consulting on EA, BPM, IS, software modeling, and MDA tool customization to tailor Modelio to specific contexts and organizations. Modelio is based on Objecteering architecture, a modeling tool that has been at the forefront of model-based innovation and development for 20 years. (Modelio, 2015)

Objecteering was the first tool to support MDA for UML through profiling and covers the entire modeling lifecycle, including requirements management, reporting and documentation, and code generation. These experiences contributed to the development of Modelio, enabling the production of models, code and documentation. Modelio provides complete coverage to support modeling activities for each stakeholder within a company. Its modeling support includes UML2, BPMN, Enterprise architecture, SOA architectures, objective analysis, requirements (Lanusse, 2009).

In addition to a predefined set of model-driven, code-driven code generation modules, Modelio provides a strong integrated MDA capability through its UML Profile and advanced Java API to access the metamodel, transform models and customization tools. Modelio supports distributed teamwork and development-oriented model cooperation through Subversion (SVN) integration. Flexible and efficient. Some of the features include in Modelio: model-oriented Java, C #, C ++ code generation, distributed teamwork using SVN, MDA Designer; UML profile editor, Java API for metamodel manipulation; Modeling the SOA, BPM, and BPMN Modeler architecture, generates or reverses interfaces XML (XSD) schemas and WSDL code (Defray, 2010).

3.3.3. Visual Paradigm

The Visual Paradigm (VP) for UML is a powerful visual tool for UML CASE. It is designed for a wide range of users, including software engineers, systems analysts, business analysts, and systems architects, such as those who are interested in reliably building software systems through the object-oriented approach. It supports over 20 diagram types including UML 2.1, BPMN, SysML and many others.

VP supports the software development cycle - project analysis, implementation, testing, and deployment. Helps build faster, better, and cheaper apps. You can create all types of UML diagrams and generate documentation. The tool has a good working environment, which facilitates the visualization and manipulation of the modeling project. Generate Java source code from the UML class model and let the UML model reflect the change you made to the source code. Round-trip engineering helps keep Java source code and software design synchronized. Each time you generate code or update the UML model, the changes are merged (Visual Paradigm-UML tool, 2008).

Generate executable Java code and HTML documentation to start development with the RESTful API. Backend development and client-side consumption are supported and benefited by the HTML documentation and generated visual communication model.

Support for UML profiles is provided, as well as the use of graphical notation for stereotypes. When implementing a profile, adding stereotypes already chooses the metaclass it will extend. This extension is not shown explicitly, as in the Papyrus UML2 Modeler or Modelio tools. You can also import / export templates using the format XMI model interchange standard without loss of information. Does not support distribution diagram modeling.

Design and deploy software in a single environment i.e. support various IDE such as Eclipse, Visual Studio and others by simply choosing the favorite. With the UML editor seamlessly integrated with the IDE, you can focus on developing your great software comfortably. Just click once to update your UML design code or to update your UML class model based on source code. Popular TOGAF software with an industry-exclusive TOGAF ADM lifecycle management tool and is used by the world's best-known companies. It has a complete set of backlog tools and agile management processes that make your agile projects more effective (Paradigm, 2010).

3.4. Description of Study Case

3.4.1. Study Case Selection.

The software development industry is most often driven by business opportunities. In this scenario, the ability to increase market share and reduce time to market are important issues that strengthen two aspects of software development processes, the ability to use experience and the ability to gain independence from software platforms.

These aspects are related to reuse at different levels of abstraction and raise some important questions.

• How to automate the generation of software artifacts from high level requirements?

• How to ensure the properties of the requirements throughout the development process?

To address this problem, Software Engineering professionals and organizations such as OMG have developed ways to systematize software asset creation, Model Driven Architectures (MDA) (OMG, 2014), which can be used to achieve wide reuse. scale through frameworks and configurable framework platforms independent of software asset representations. (Deelstra, Synnema and Gurp, 2003).

UML diagrams are formal representations of various software components and their stockholders; therefore, they play an important role in creating software. Therefore, the reason for choosing each case study will be explained below, and each case study will be represented by a UML diagram. It was taken from uml-diagrams.org.

The Hospital Management case study is represented by class diagram. This case study has a medium level of compliance because all elements of this diagram are encompassed in this case study. By specifying further, we can see that this diagram has all sorts of attributes, and we can also see that the link between classes encompasses all the types that the class diagram contains, namely aggregation, association. This case study was chosen because it ensures that in the generation of the code it will cover all the elements of the class diagram. Another reason for choosing this case was the easy semantic understanding of the case, that is, we can easily understand the statistical structure of the system, that when the diagram will be generated from the code, we can easily identify if all the elements were covered.

As mentioned earlier all case studies will be represented by a diagram, in this case they will not be an exception, as the Online Shopping case study is represented by activity diagram. As noted in the previous case, one of the reasons for choosing is due to the case study covering the largest feature number of the activity diagram. Apart from having an easy understanding, another reason was that this diagram has a direct impact on the application, and it is important to apply benchmarketing to this case.

By observing this Bank ATM case study, which is represented by state diagrams, we can see that it has a high complexity, as it covers all the characteristics of the state diagram, and in addition we also find that it contains a composite state. and actions. Considering the characteristics, we can say that it will be a case like a real one. Given that the purpose of our study is to recommend the best round-trip engineering tool, this case will be a good idea.

The component diagramming that represents the components of a system will be the next diagram that will represent the last case study, Sentinel HASP Licensing Components. This case is of great importance in the system, since each component of this diagram represents a part of the system. This case is a case of medium complexity, as it only covers all the characteristics of the component diagram, namely components, artifacts and interface that serves to communicate with each component, and the aggregation and association links. Apart from the features, another reason to choose is to be able to map with object-oriented languages.

3.4.2. Study Cases

The case that will be used in this example is a hospital domain model diagram (Hospital Management, 2016). The domain model for the Hospital Management System is represented by several class diagrams. The purpose of the diagram is to show and explain the hospital structure, staff, patient relationship, and patient treatment terminology. In the figure below, a "Person" may be associated with different hospitals, and a "Hospital" may employ or serve several "People". The "Person" class derived the name and "homeAddress" attributes. The name represents the full name and can be combined from the title, first name (or first), middle name and family name (or last name). The patient class derived the age of the attribute that can be calculated based on their date of birth and current date or date of hospitalization. The "Patient" class inherits attributes from the "Person" class. Several inherited attributes, name, gender, and "date of birth" are shown with the caret symbol.

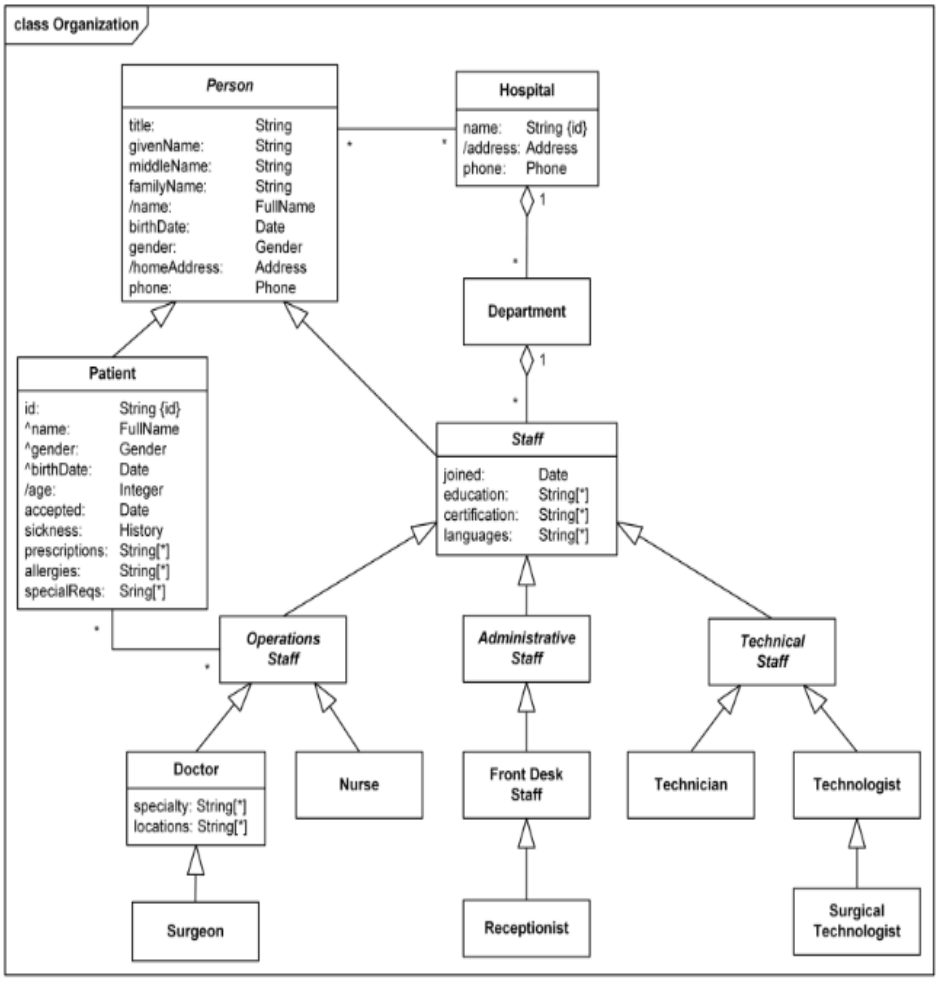

Figure 11 – Class Diagram

It is a case study (Bank ATM. 2016) of a UML behavioral state machine diagram showing the top-level ATM state machine. In the initial state, the ATM awaits customer interaction. The ATM state changes from Idle to Serving Customer when the customer enters the bank or credit card into the ATM card reader. Upon entering the Serving Customer state, the readCard input action is performed. The Customer Service state is a state composed of sequential substances Customer Authentication, Transaction Selection, and Transaction. The customer service state returns to the inactive state after the transaction is completed. The state also has the ejectCard exit action, which frees the customer card when leaving the state, regardless of what caused the transition out of state.

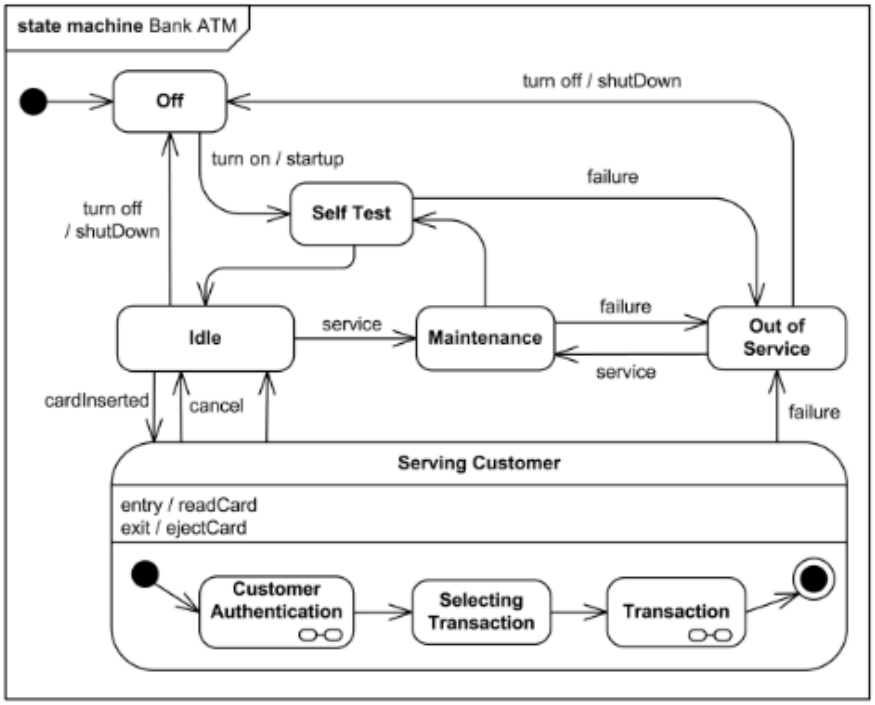

Figure 12 – State Machine

This is a case study (Online Shopping, 2016) of the activity diagram for online shopping. The online customer can browse or search items, view specific items, add them to the shopping cart, view and update the shopping cart, complete the purchase. The user can view the shopping cart at any time. Checkout is assumed to include user registration and login. This example does not use partitions, it is assumed that most actions are performed by the online client.

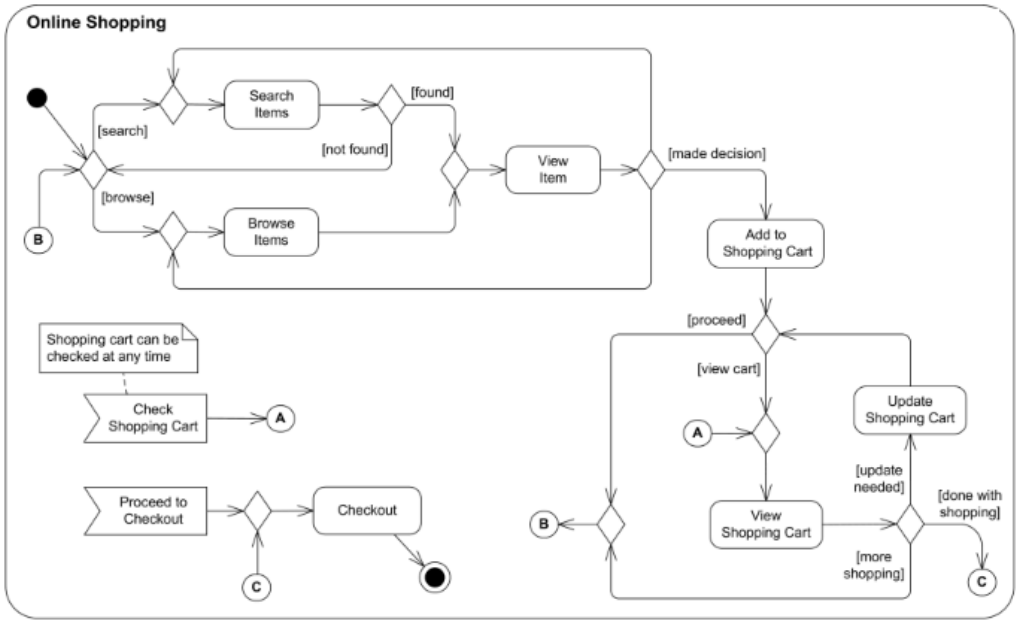

Figure 13 – Activity Diagram

This case study (Sentinel HASP Licensing Components, 2016) is a UML component diagram with a simplified view of the components delivered and deployed using the SentNet SafeNet Sentinel HASP software security solution and the licensing API. At the top of the diagram we have some software implemented using the Sentinel HASP - .Net License Status application and the License Services Java component. The License Status application is for showing license status and is manifested (implemented) by the license_status.exe artifact. The Java component of License Services implements the License Service interface and may be used by other Java applications or services. The License Status application uses the Net component of License Services through the License Service interface implemented by this component. The License Services Net component uses the HASP .Net API provided by the HASP .Net Runtime component, which is part of the Sentinel HASP product. The License Services Java component uses HASP Java Native Interface Proxy to communicate with the HASP Java Native Interface component, both components provided by Sentinel. When the product is used on Microsoft Windows, HASP Java Native Interface may manifest as HASPJava.dll (32-bit OS), HASPJava_x64.dll or HASPJava_ia64.dll (64-bit OS).

Figure 14 – Component Diagram

In the next chapter, a brief discussion, of the three-case method is presented, namely: direct engineering, reverse engineering, and round-trip engineering, used for experimental validation of the tools described above. Case study diagrams were created using previously chosen UML case tools that automatically generate Java code from UML diagrams.

3.5. Description of Techniques

3.5.1. Forward engineering

Starting by explaining the case study whose objective is to apply the Forward Engineering method in several UML diagrams using the chosen frameworks and make the performance study of each one or it is already chosen and criteria that were previously chosen to analyze the code that was generated. for each framework, applied to each diagram. That said we start by implementing the diagrams in UML 2.5 in the editor of each tool. Having the diagrams, you start by setting up the code so that you can generate the code, or you start creating the profiles and context for each element of the diagram. After everything is prepared, the code is generated. Finished the code generation, through the metrics framework (Metrics, 2015) we do a quantitative analysis of the code and then a qualitative analysis to ensure that no unnecessary code was generated or without context. Having all the necessary data in the next chapter will be made a detailed analysis of the collected data.

3.5.2. Reverse engineering

Like the previous case study, but with the objective of applying the reverse engineering method in code of several components of an application and generate the UML diagrams used in the previous frameworks and make the performance study of each one or it will be and criteria's that predefined to make analysis to the elements of each diagram. That said, having already generated the code from the previous experiment, and having already defined the profiles and context, the diagrams were generated. Having everything ready we applied the same framework (Metrics, 2015) to generate the metrics and then do quantitative analysis of them, also analyzed the diagrams to understand if the context has not been lost or make a qualitative analysis of the diagrams. And as in the previous case the collected data of experiment will be analyzed later.

3.5.3. Round-trip engineering

Given the previoun case studie, where the aim was to apply the forward and reverse engineering methods to this case study, round-trip engineering will be applied, which is a method of keeping diagrams and code up to date, or when developed. a new requirement is that the current diagram does not reflect reality and for that to be corrected and synchronized the diagrams automatically. That said having the code and already having diagrams, to be able to test was made changes in the code and made synchronization and later changed the diagram and made the synchronization. Having the changes made and applied as in the previous case the framework where will be generated the metrics and make analysis as quantitative with qualitative**.**

3.6. Application of Scenario

3.6.1. Scenario forward and reverse engineering without any changes

The following scenario explains how the experiment was performed. For this reason, we will start by installing the necessary tools to later prepare the environment to be able to pass the experiment. The experiment itself begins with drawing the diagrams, which were referred to in the case studies. After the diagram will be validated with the validation framework of the UML modeling tool itself. Having the diagram drawn, we proceeded to code generation i.e. Forward Engineering. Considering the code that was generated, we applied the metric count and validated them. After validating the diagram, we reverse engine, and repeat the same process as in forward engineering.

3.6.2. Scenario forward and reverse rngineering with changes

In this new scenario will be some changes in the scenario that was applied later, namely by adding new elements to the diagram. In the case study Hospital Management will be added a new class named "Reclamation", later it will have 6 attributes, id, complaint, person, staff, date. This class will be linked by a multiple association with the class "Staff". In the Bank ATM case study, the composite state "Serving Costumer" will be modified and a new state will be added which will be linked "Transaction" and the final state. In the Online Shopping case study, a "Remove from Shopping Chart" activity will be added and linked the decision figure before the "Add to Shopping Chart" activity. In the last case study, a new Component will be added called "Java Library, with generic artifacts. This new component will be linked to" Java native Interface ". After fetal changes, code generation will be done i.e. forward engineering. Considering the code that

was generated, we apply the count We repeat the same process as in forward engineering, but apply it to code and then do reverse engineering, Scenario round-trip engineering with changes

3.6.3. Scenario round-trip engineering with changes

Applying the same procedure that was applied in the previous scenario, namely to make alteration to the original cases. But what will differ in this scenario is the methods that will be used or will be applied to round-trip engineering. This method will be applied when there are changes in diagrams and they will have to be updated as new functions, but in the same diagram. The same will be applied to the code, later the round-trip engineering will be applied. Having the results will be applied to the metric count.

Chapter 4 – Analysis and Discussion of Results

4.1. Comparison Criteria

We restrict ourselves to applying relatively simple metrics to collect from source code, as source code is often the most reliable source of information. Metrics (Metrics, 2015), which is an object-oriented language metrics framework, was chosen. From the broad set of possible metrics, we select design metrics, i.e. metrics that can be extracted from the source code itself. These metrics are generally used to evaluate the size and, in some cases, the quality and complexity of the software. The metrics we use are called direct measurement metrics because their calculation involves no other entities (Fenton, Pfleeger, 1996). It is not use indirect measurement where metrics are combined to generate new ones, because the presented use case measurement works best with direct measurements. Examples of indirect metrics include programmer productivity, defect detection density, or defect density module as well as more code-oriented (Chidamber, Kemerer, 1994). We chose to use metrics that can be extracted from code entities and have a simple and clear definition. As such, we do not use composite metrics, which raise the question of consistency dimension. In Table 1, we list all the metrics mentioned in this experiment.

Table 1 – Metrics

Metrics	Acronyms	Class Diagram	Component Diagram	State-machine Diagram	Activity Diagram
McCabe Cyclomatic Complexity	VG	X	X	X	X
Number of parameter	PAR	X	X	X	X
Next Block Deep	NBD	X	X	X	X
Affered Coupling	CA	X	X	X	X
Effered Coupling	CE	X	X	X	X
Instability	RMI	X	X	X	X
Abstractness	RMA	X	X	X	X
Normalized Disntance	RMD	X	X	X	X
Depth of Inheritance Tree	DIT	X	X	X	X
Weighted methods per Class	WMC	✓	✓	✓	✓
Number of Children	NSC	✓	✓	✓	✓
Number of Overridden Methods	NORM	✓	✓	✓	✓
Lack of Cohesion of Methods	LCOM	X	X	X	X
Number of Attributes	NOF	✓	✓	✓	✓
Number of Static Attributes	NSF	✓	✓	✓	✓
Number of Methods	NOM	✓	✓	✓	✓
Number of Static Methods	NSM	✓	✓	✓	✓
Specialization Index	SIX	X	X	X	X
Number of Class	NOC	✓	✓	✓	✓
Number of Interface	NOI	✓	✓	✓	✓
Number of Packeges	NOP	✓	✓	✓	✓
Total Lines of Code	TLOC	✓	✓	✓	✓
Methods Lines of Code	MLOC	✓	✓	✓	✓

4.2. Analysis and Discussion of Results

4.2.1. Applying Scenarios to Papyrus

Table 2 shows a summary table of the Forward Engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of provides the complete implementation of the Hospital Management System class diagram using the Papyrus framework.

Table 2 – Results class diagram forward engineering

Class Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	11	11
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	28	28
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	54	54
Number of Static Methods	NSM	0	0
Number of Class	NOC	21	21
Number of Interface	NOI	0	0
Number of Packeges	NOP	1	1

The case study helps us evaluate the code generated from the code generation tool. Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated Java code skeleton is built from the UML class diagram, which keeps the complexity of the generated code low because no extraneous classes are created. The methods have a sturdy equivalent to the class diagram. The code in general does not present errors, because it was compiled, and all libraries were imported as configured in UML. The code came commented and well documented, because it is clear to identify what the method is supposed to do and what each attribute means.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the class diagram was generated 230 lines of code, from these lines were accounted 21 classes, 54 methods, 27 attributes and 11 subclasses (i.e. a class that implements an account interface as a direct child of this interface). Deepening this analysis, and understanding what these numbers tell us, we must look at the class diagram and see if they reflect what is on the diagram. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics is a reference metrics and to ensure that this number makes sense we have to analyze the other metrics because there is a dependence between them.

By analyzing the number of classes in the diagram and the class number of the code we find that quantitatively they are the same, but also, we find that the descriptions are identical so we can conclude that the classes were generated as expected. By moving to the following metrics, numbers of the methods and performing the same analysis that was done before we can conclude that quantitatively is correct because all the methods that have been rotated derive from the attributes, to speak more concretely an attribute will generate two methods Set_attribute and Get_attribute, taking into account. Since we have 27 attributes, we can conclude that the same number of methods were generated equivalent to number of attributes. Speaking of attributes as we saw 27 attributes were generated and, in the diagram, there were 27 so we can also conclude that the code generation went as expected. We can also state that 11 subclasses were generated i.e. a class that implements another class counts as a direct child of that class and analyzing the diagram we conclude that the number coincides. This analysis can also be verified in the graph below, where we can see that the total theoretical area covers the practical area in totality, being the theoretical total that is referred to the class diagram and the generated code practice.

Table 3 shows a Reverse Engineering use case summary table showing the results of generating the UML diagram from code. The class diagram consists of providing the complete implementation from the Hospital Management System Java code using the Papyrus framework.

Table 3 – Results class diagram reverse engineering

Class Ciagram			
Metrics	Acronyms	Total Real	Total Teorical
Number of Children	NSC	11	11
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	28	28
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	54	54
Number of Static Methods	NSM	0	0
Number of Interface	NOI	0	0
Number of Interface	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can

see that the classes were generated correctly, i.e. have the same type, the attributes were also generated as expected having the correct types of them, but in some cases having some attributes repeated. Associations in some classes were generated correctly, but in some sense, it was reversed, i.e. instead of having A-to-B membership it was the other way around but in general associations are present. Generalizations are also present and analyzed the code, the previous diagram we can conclude that were generated as expected. Checking the diagram in general is an eligible diagram, structurally well accomplished.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 21 classes, 54 methods, 27 attributes, 4 associations and 13 aggregations. By proceeding to make a more detailed comparison of the metrics between the generated diagram and the code, we find that the number of diagram classes and the class number of the code are quantitatively the same, but we also find that the descriptions and typology are the same so we can conclude that the classes were generated as expected.

By moving to the following metrics, numbers of the methods and performing the same analysis that was done before we can conclude that quantitatively is correct because all the methods that were rotated derive from the attributes i.e. the GETTERS and SETTERS where 57 methods were generated that reflects the number. Attributes which are 27. When checking the number of bindings that are the getters and the attribute setters that help in the binding of the classes, they are correct but as explained earlier they are exchanged. Verifying the same were the aggregations we can say that they at the quantitative level are equal to the diagram and correctly diffused. In this way we can conclude that Papyrus in general got the general diagram that is structurally well done. This analysis can also be verified in the graph below, where we can see that the total theoretical area covers the practical area in totality, being the theoretical total that is referred to the class diagram and the generated code practice.

By analyzing the third case study, round-trip engineering, we found that changes were made to the previously constructed class diagram. The changes that were made was to add a new class with 3 attributes and 2 methods and was linked to class "Person", this change was made to see the behavior of the tool when changing the structure of the diagram. It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged but one more class with its attributes and methods was added. Therefore, we can conclude that Papyrus can easily handle changes

to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively updating the diagram it was found that the diagram was updated with the new class keeping what was unchanged. Having this analysis, we conclude that for the class diagram, Papyrus can handle Round-Trip Engineering.

Table 4 show a summary of the Forward Engineering use case showing the results of code generation from UML component diagram. The generated Java code consists of provides the complete implementation of the Sentinel HASP Licensing Components system diagram using the Papyrus framework. The case study helps us evaluate the code generated from the code generation tool. Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated Java code skeleton is built from the component diagram corresponds to reality. The code in general do not present errors, because it was eaten. The code came well documented, since it is clearly possible to identify the components and their benefits as well as the interfaces and their methods.

Table 4 – Results component diagram forward engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	0	0
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	12	12
Number of Static Methods	NSM	0	0
Number of Class	NOC	5	5
Number of Interface	NOI	9	9
Number of Packeges	NOP	1	1

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the activity diagram was generated 297 lines of code, of these lines were accounted 5 classes, 12 methods, 9 interfaces and 1 package. In deepening this analysis to understand what these numbers tell us we must look at the component diagram and understand if they reflect what is in the code.

Before proceeding to make a more detailed comparison of metrics we can observe that the metrics of number of lines is not a metrics that can be observed directly in the diagram and to verify if this number makes sense, we must analyze the other metrics because there is a dependence between, they. Looking at the component diagram, we can see that the

system has 9 components that provide and consume behaviors through interfaces. We can also affirm a component contains one or more artifacts, and each artifact can be composed of class, methods and attributes but in the context of our system we can observe that the components contain information that cannot be accounted for only by making assumptions.

Seeing the 5 classes that were generated and analyzing the diagram we conclude that they were generated with the structure of the Java project, where was created Service and Controller classes where will be made the logic and called the components that contain the artifacts. For this reason, we cannot make a direct relationship with the diagram. By checking the methods that were created we can say that they were generated due to the existing operations between components and interfaces. By checking the number of them we can say that they reflect what is in the diagram. The 9 interfaces that were created in the code are for component interconnection, and we can verify the same by looking at the diagram.

Table 5 shows a summary table of the Reverse Engineering use case that shows the results of UML diagram generation from code. The component diagram consists of providing the complete implementation from Sentinel HASP Licensing Components system Java code using the Papyrus framework.

Table 5 – Results component diagram reverse engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	0	0
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	12	12
Number of Static Methods	NSM	0	0
Number of Interface	NOI	9	9
Number of Packeges	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can observe that the generated components are in conformity with the code, because it can be clearly identified in the diagram, as well as their artifacts. We can also observe that the

interfaces were generated correctly, i.e. each interface is correctly associated with each component, we can also observe that the

Associations between interfaces were generated, but there were 3 cases where they did not reflect the code, because there are interfaces where it is consuming behaviors and the goal was to provide. Generalizations are also present and analyzing the code we can gather that were generated in the diagram as expected. Checking the diagram in general is an eligible diagram, structurally well accomplished.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 9 components, 10 interfaces, 16 associations and 2 aggregations. By making a more detailed comparison of the metrics between the diagram and the code, we observe that in the diagram there are 9 components, and analyzing the code we identify the same number of components so we can say that the tool can clearly identify in the code the component and Reflecting them in the diagram, we also observed that all the artifacts contained in the component were also undetected. Attention to the artifacts that were used in the case were generic artifacts, since the aim of the study is to show that the tool can identify them and transpose them in the diagram. Thus, we can clearly state that the components and their artifacts were generated in accordance with the defined code and rules. Observing the diagram it is possible to verify that 10 interfaces were generated and knowing the context of the system in which the interfaces are used to communicate with the components we verify that each component has an interface that is quantitatively correct, there is a component that has two interfaces one that consumes information i.e. where the system receives information and other interfaces where it communicates with the other components. That said we can verify that the tool handles interfaces well in context of the component diagram. A commenting point has been configured to appear the view of the miniature interfaces to be easier to manipulate, but this did not happen, and intervention was needed to put in the desired view but beware this point does not interfere in its validity or functioning.

The associations generated between the components quantitatively are correct but as mentioned above there were inconsistencies. Verifying the same were the aggregations we can say that they at the quantitative level are equal to the diagram and correctly diffused. Thus, we can conclude that Papyrus overall got the diagram as intended, the inconsistencies that exist can have a big impact if not corrected, so it is a point to improve.

By analyzing the third case study, then is Round-Trip Engineering we found that changes were made to the previously constructed component diagram. The changes that were made was to add a new component with 2 artifacts and was connected to the "License Services" interface, this change was later made to see the behavior of the tool when changing the structure of the diagram. It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged but one more component was added. the respective class and method that connects the interface. Therefore, we can conclude that Papyrus can easily handle changes to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively updating the diagram it was found that the diagram was updated with a new component but keeping what was unchanged. Having this analysis, we conclude that for the class diagram, Papyrus can handle Round Trip Engineering.

Table 6 shows a forward engineering use case summary table showing the results of code generation from the statechart diagram. The generated code consists of providing the complete implementation of the Bank ATM system using the Papyrus framework, the choice of C++ language for this case was because it has the same paradigm as Java i.e. it is an object-oriented language, and you can make a parse equivalent to what was being done in Java. The reason for chose C++ for the statechart diagram was because Papyrus does not support Java for this diagram currently. The other tools that have been chosen also support C++, so it will be a valid comparison.

Table 6 – Results state-machine diagram forward engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	24	26
Number of Static Attributes	NSF	4	6
Number of Methods	NOM	61	65
Number of Static Methods	NSM	1	1
Number of Class	NOC	16	18
Number of Interface	NOI		
Number of Packeges	NOP	1	1

The case study helps us evaluate the code generated from the code generation tool. Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated code skeleton is built from the UML statechart diagram, which

maintains the complexity of the generated code, as various methods are created where various operations for state evolution and validation are implemented. The code in general does not present errors, because it was compiled, and all libraries were imported as configured in UML. The code came commented and well documented, because it is possible to clearly identify the methods and attributes.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the use case diagram was generated 594 lines of code, from these lines were accounted 16 classes, 62 methods of which one is static, 22 attributes and 4 static attributes, 6 subclasses. Deepening this analysis, and understanding what these numbers tell us, we must look at the case use diagram and see if the code reflects what is in the diagram. Before proceeding to make a more detailed analysis of metrics, we can verify that the line number metrics is a reference metrics and to ensure that this number makes sense we have to analyze the other metrics because there is a dependency between them.

Starting with the first metric number of classes that exist in the code reflects the connections between the states of the diagram and as we can see that there are 16 classes as well as there are 16 connections, then we can say that the tool succeeded in generating the classes. For the number of methods we can verify in the code that for each state was created a method, as well as we verify that for connection was also created a method we also found that methods were created for the attributes i.e. when we talk about methods for example we can consider that each state is an attribute and for each attribute two methods were generated, for the state evade was i.e. the state transaction was also generated, after analyzing the diagram we can conclude that there is consistency between the two. The attributes that were generated are consistent with the diagram we can observe that observe through the code and diagram, that is in the state diagram a state can have several state for example in our case as we can observe the state "Self-Test" has several state as per example "Failure" that all states will be reflected in attributes. Thus, we can conclude that the number of attributes that was generated corresponds to what was drawn in the diagram. There are 6 subclasses and they were generated because there are states that have sub stated, checking the diagram we can conclude this. There are 4 statistical attributes that represent the initial and final states of the system. After qualitative and quantitative analysis, we conclude that the code generation was performed as expected.

Table 7 show a Reverse Engineering use case summary table that shows the results of code generation from the statechart diagram. The generated diagram consists of providing the complete implementation of the Bank ATM system using the Papyrus framework. The generation of the diagram was made from the C ++ language and choosing this language for this specific diagram was explained earlier.

Table 7 – Results state-machine diagram reverse engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	24	24
Number of Static Attributes	NSF	4	4
Number of Methods	NOM	61	61
Number of Static Methods	NSM	1	1
Number of Interface	NOI		
Number of Packeges	NOP	1	1

Checking the diagram, we can see that the states of the statechart diagram that were generated are code compliant. To validate the following we find that in the diagram are present the states and their composite states, which are well identified. Quantitatively we can see that 8 states were generated and a complex state and 3 static states that are the initial state and the final state, say static because they never change. Since validation cannot be done directly since a state in the code is reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram. Next, we must analyze the transitions between states. For this it was done as previously an interpretation of the code and analysis of the diagram and the conclusion was reached that there are 2 transitions that are with the changed directions, i.e. one the "Cancel" transition should leave the composite state and enter the "Idle state". "and the transition "cardInserted "should leave the state" Idle "and enter the composite state, in the analysis of the code concluded that this error happened because there was another unnamed transition and the conflict arose changing the meaning. 19 connections were generated, observing the diagram we can see that all the states are connected properly, which reflects what is in the code. We can conclude that the Papyrus tool in general managed to generate the statechart diagram as intended, the existing inconsistencies were corrected manually,

but we can verify that tool does not deal well with transitions, to make the errors of this kind meticulous we must have a code that follows the standards of good practice.

By analyzing the third case study, i.e. Round-Trip Engineering we found that changes were made to the previously constructed statechart diagram. The changes that were made was to add a new state "Wait Transaction" within the composite state. It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged, but the new changes were added. Therefore, we can conclude that Papyrus can easily handle changes to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively, updating the diagram it was found that the diagram was updated keeping the state unchanged, I add the state but the respective transaction that diverged to link to the remaining state. it was not added, but the sense of transitions within the composite state was changed. Obviously, it is difficult to understand in detail how Papyrus determined the rules for generating these transitions, as it is an internal tool error. After analysis we conclude that applying Round Trip Engineering to the statechart diagram, the diagram is updated successfully but with some inconsistencies, being critical to the diagram operation.

Table 8 how's a summary table of the Forward Engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of provides the complete implementation of the Online Shopping system activity diagram using the Papyrus framework.

Table 8 – Results activity diagram forward engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	20	22
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	54	60
Number of Static Methods	NSM	0	0
Number of Class	NOC	12	14
Number of Interface	NOI		
Number of Packeges	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the diagram. The generated Java code skeleton is constructed from the activity diagram, which

maintains a medium level complexity of the generated code, bearing in mind that classes are created with methods where validation and control operations are implemented. The code in general do not present errors, because it was eaten. The code came well documented, because it is easy to identify the methods and what each attribute means.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the activity diagram was generated 522 lines of code, of these lines were accounted 12 classes, 54 methods, 20 attributes and 6 subclasses. Deepening this analysis and understanding what these numbers tell us we must look at the activity diagram and understand if they reflect what is in the code. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics with us above to ensure that this number makes sense we have to analyze the other metrics because there is a dependency between them.

Starting with the first metric number of classes that exist in the code relate the activities directly, i.e. the activity will give rise to one or more actions of an object, we can also say that a class here reflects the object, checking the diagram we can observe that there are 7 activities and assuming that in the current context each activity will reflect an object and also assuming that for object will have associated at least one class we can say that will be generated at least 7 classes, we can also observe that in the activity diagram there are Other elements such as starting points, decision points and assuming that the structure of a Java project, should be created a Service and Controller class where the logic will be done.

Considering this analysis of the diagram and the generated code we find that the generated class number is a reality number, thus concluding that the tool succeeded in generating the classes. Moving on to the number of methods we can see that in the diagram there is activity that will give rise to action as we previously thought. But as each action will be reflected in the code, to answer this question I must realize that an action in the application will be associated with an attribute, and for attribute we can generate n number of methods. Seeing the current context of the diagram we can see that for each activity there will be more than one action, analyzing the code we can see that what is in the diagram is reflected in the code or we can conclude that there is consistency between the two. Given all the previous analysis and current context of the diagram we realize that the number of attributes is related to the actions, so the number of attributes generated is

reflected in the diagram, so we conclude that the tool handles the attributes associated with a diagram well. of activity.

Speaking of subclasses, which are 6, we can say that they were rotated in the context of the need to take advantage of certain characteristics of another class, that is, there are classes that inherit characteristics of another, speaking of this we can verify that there are actions that have Similar characteristics As soon as the existence of subclass is normal, looking at the context beforehand, we can say that the number of subclass reflects what is in the diagram.

Table 9 show a Reverse Engineering use case summary table showing activity diagram that was generated from that of the Java code. The generated diagram consists of providing the complete implementation of the Online Shopping system using the Papyrus framework.

Table 9 – Results activity diagram reverse engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	24	20
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	58	54
Number of Static Methods	NSM	0	0
Number of Interface	NOI		
Number of Packeges	NOP	1	1

Checking the diagram, we can see that the activities of the activity diagram that were generated code and compiled. To validate the following we verify that in the diagram the activities that are well identified, as well as checking the pre-condition and post-condition duly marked at the beginning of each cycle of actions and at the end of them. Quantitatively we can see that 7 activities were generated. Since validation cannot be done directly because each activity is formed by various actions state in the code and reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram.

Next, we must analyze the transitions between the activities. For this as previously done, we will have to do an interpretation of the code and analysis of the diagram. After

the analysis it was concluded that they are correctly generated, each activity is linked with previous one, we also verify that there are moments of decision, and when checking the code, they are correctly connected. Given that these numbers cannot be directly accounted for in the code, we can say that 12 classes, 54 methods, 20 attributes and 6 sub-classes gave rise to 24 transitions and 9 decision moments, observing the diagram we can see that all states are connected correctly.

Qualitatively and qualitatively the Papyrus tool did a good job, as I was able to generate the diagram completely without errors, so we can conclude that Papyrus was able to generate the activity diagram as intended.

By analyzing the third case study, i.e. Round-Trip Engineering, we found that changes were made to the previously constructed activity diagram. The changes that were made was to add a new activity "Remove from Shopping Chart". It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged, but the new changes were added. Therefore, we can conclude that Papyrus can easily handle changes to its long-life cycle diagram. After making the change in the code, where it was added the same as was added in the diagram, respectively, updating the diagram it was found that the diagram was updated keeping the activities unchanged but adding the new activity but its link that diverged to link the state has not been added and removing the link between the "Add to Shopping Chart" activity. Being an internal tool error, it is difficult to understand in detail how Papyrus determined the rules to remove the following links. After analysis we conclude that by applying Round Trip Engineering to the activity diagram, it updates the diagram successfully but with some inconsistencies, which are critical for the diagram operation.

4.2.2. Applying Scenarios to Modelio

Table 10 shows a summary table of the Forward Engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of provides the complete implementation of the Hospital Management System class diagram using the Modelio framework.

Table 10 – Results class diagram forward engineering

Class Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	11	11
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	27	27
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	56	54
Number of Static Methods	NSM	0	0
Number of Class	NOC	21	21
Number of Interface	NOI	0	0
Number of Packeges	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated Java code skeleton is built from the UML class diagram, which keeps the complexity of the generated code low because no extraneous classes are created. The methods have a sturdy equivalent to the class diagram. The code in general do not present errors, because it was eaten. The code was documented, as it is possible to identify the attributes methods easily.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the class diagram was generated 242 lines of code, from these lines were accounted 21 classes, 56 methods, 27 attributes and 11 subclasses. Deepening this analysis, and understanding what these numbers tell us, we must look at the class diagram and see if they reflect what is on the diagram. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics is a reference metrics and to ensure that this number makes sense we have to analyze the other metrics because there is a dependence between them. By analyzing the number of classes in the diagram and the class number of the code we find that quantitatively they are the same, but also, we find that the descriptions are identical so we can conclude that the classes were generated as expected.

By moving to the following metrics, numbers of the methods and performing the same analysis that was done before we can conclude that quantitatively is correct because all the methods that have been rotated derive from the attributes, to speak more concretely an attribute will generate two methods Set_atribute and Get_attribute, taking into account Since we have 27 attributes we can conclude that the same number of methods were

generated equivalent to number of attributes. Speaking of attributes as we saw 27 attributes were generated and, in the diagram, there were 27 so we can also conclude that the code generation went as expected. We can also state that 11 subclasses were generated i.e. a class that implements another class counts as a direct child of that class and analyzing the diagram we conclude that the number coincides. This analysis can also be verified in the graph below, where we can see that the total theoretical area covers the practical area in totality, being the theoretical total that is referred to the class diagram and the generated code practice.

Table 11 shows a Reverse Engineering use case summary table showing the results of generating the UML diagram from code. The class diagram consists of providing the complete implementation from the Hospital Management System Java code using the Modelio framework.

Table 11 – Results class diagram reverse engineering

Class Ciagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	11	11
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	28	28
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	54	54
Number of Static Methods	NSM	0	0
Number of Class	NOC	21	21
Number of Interface	NOI	0	0
Number of Interface	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can see that the classes were generated correctly, i.e. have the same type, the attributes were also generated as expected having the correct types of them, but in some cases having some attributes repeated. Associations in some classes were generated correctly and associations are generally present. Generalizations are also present and analyzed the code, the previous diagram we can conclude that were generated as expected. Checking the diagram in general is an eligible diagram, structurally well accomplished.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 21 classes, 56 methods, 27

attributes, 4 associations and 13 aggregations. In making a more detailed comparison of the metrics between the generated diagram and the code, we find that the number of diagram classes and the class number of the code are quantitatively the same, but we also find that the descriptions and typology.

By moving to the following metrics, method numbers and performing the same analysis that was done before we can conclude that quantitatively is correct because all the methods that were rotated derive from the attributes i.e. the GETTERS and SETTERS where 56 methods were generated that reflects the number. Attributes which are 27. When checking the number of bindings which are the getters and the attribute setters that help in the binding of the classes, they are correct but as explained earlier they are exchanged. Verifying the same were the aggregations we can say that they at the quantitative level are equal to the diagram and correctly diffused. Thus, we can conclude that Modelio in general got the diagram that is structurally well done.

By analyzing the third case study, i.e. Round-Trip Engineering we found that changes were made to the previously constructed class diagram. The changes that were made was to add a new class with 3 attributes and 2 methods and was linked to class "Person", this change was made to see the behavior of the tool when changing the structure of the diagram. It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged but one more class with its attributes and methods was added. Therefore, we can conclude that Modelio can easily handle changes to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively updating the diagram it was found that the diagram was updated with the new class keeping what was unchanged. Having this analysis, we conclude that for the class diagram, Modelio can handle Round Trip Engineering.

Table 12 shows a summary table of the Forward Engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of providing the complete implementation of the Sentinel HASP Licensing Components system diagram using the Modelio framework.

Table 12 – Results component diagram forward engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	0	0
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	14	12
Number of Static Methods	NSM	0	0
Number of Class	NOC	5	5
Number of Interface	NOI	9	9
Number of Packages	NOP	1	1

The case study helps us evaluate the code generated from the code generation tool. Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated code structure corresponds to the diagram. The code in general do not present errors, because it was eaten. The code came well documented, since it is clear to identify the components, their artifacts as well as the interfaces and their methods.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the activity diagram 305 lines of code were generated, from these lines were accounted 5 classes, 14 methods, 9 interfaces and 1 package. In deepening this analysis to understand what these numbers tell us we must look at the component diagram and understand if they reflect what is in the code.

Before proceeding to make a more detailed comparison of metrics we can observe that the metrics of number of lines is not a metrics that can be observed directly in the diagram and to verify if this number makes sense, we must analyze the other metrics because there is a dependence between, they.

Looking at the component diagram, we can see that the system has 9 components that provide and consume behaviors through interfaces. We can also affirm a component contains one or more artifacts, and each artifact can be composed of class, methods and attributes but in the context of our system we can observe that the components contain information that cannot be accounted for only by making assumptions. Seeing the 5 classes that were generated and analyzing the diagram we conclude that they were generated with the structure of the Java project, where was created Service and Controller classes where will be made the logic and called the components that contain the artifacts.

For this reason, we cannot make a direct relationship with the diagram. By checking the methods that were created we can say that they were generated due to the existing operations between components and interfaces. By checking the number of them we can say that they reflect what is in the diagram. The 9 interfaces that were created in the code are for component interconnection, and we can verify the same by looking at the diagram.

Table 13 shows a summary table of the Reverse Engineering use case that shows the results of UML diagram generation from code. The component diagram consists of providing the complete implementation from Sentinel HASP Licensing Components system Java code using the Modelio framework.

Table 13 – Results class diagram reverse engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	5	5
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	14	14
Number of Static Methods	NSM	0	0
Number of Class	NOC	5	5
Number of Interface	NOI	9	9
Number of Packages	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can observe that the generated components are in conformity with the code, because it can be clearly identified in the diagram, as well as their artifacts. We can also observe that the interfaces were generated correctly, i.e. each interface is correctly associated with each component, we can also observe that the. Associations between interfaces were generated, but there were 5 cases where they did not reflect the code, because there are interfaces where it is consuming behaviors and the goal was to provide. Generalizations are also present and analyzing the code we can gather that were generated in the diagram as expected.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 9 components, 10 interfaces, 16

associations and 2 aggregations. By analyzing in more detail the metrics between the diagram and the code, we observe that in the diagram there are 9 components and analyzing the code we identify the same number of components so we can say that the tool can identify the code in the component, we also observed that all the artifacts contained In the component were also undetected, the landfills that were used in the case are generic artifacts with no Papyrus.

Observing the diagram it is possible to verify that 10 interfaces were generated and knowing the context of the system in which the interfaces are used to communicate with the components we verify that each component has an interface that is quantitatively correct, there is a component that has two interfaces one that consumes information i.e. where the system receives information and other interfaces where it communicates with the other components. That said we can verify that the tool handles interfaces well in context of the component diagram.

The associations generated between the components quantitatively are correct but as mentioned above there were inconsistencies. Verifying the same were the aggregations we can say that they at the quantitative level are equal to the diagram and correctly diffused. Thus, we can conclude that Papyrus overall got the diagram as intended, the inconsistencies that exist can have a big impact if not corrected, so it is a point to improve.

By analyzing the third case study, i.e. round-trip engineering we found that changes were made to the previously constructed component diagram. The changes that were made was to add a new component with 2 artifacts and was connected to the "License Services" interface, this change was later made to see the behavior of the tool when changing the structure of the diagram. It was found that the code generated after this change reflects what is in the diagram then is the old code was unchanged but one more component was added. the respective class and method that connects the interface. Therefore, we can conclude that Modelio can easily handle changes to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively updating the diagram it was found that the diagram was updated with a new component but keeping what was unchanged. Having this analysis, we conclude that for the class diagram, Modelio can handle round-trip engineering.

Table 14 shows a forward engineering use case summary table showing the results of code generation from the statechart diagram. The code generated consists of providing

the complete implementation of the Bank ATM system using the Modelio framework, the choice of C ++ language for this case was because it has the same paradigm as C ++ i.e. it is an object-oriented language, and you can do it. an equivalent analysis that was being done in C ++. The reason for doing C generation for the use case diagram was because Papyrus does not support Java for this diagram currently. The other tools that were chosen also support C ++, so it will be a valid comparison.

Table 14 – Results state machine diagram forward engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	7
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	24	24
Number of Static Attributes	NSF	4	6
Number of Methods	NOM	61	63
Number of Static Methods	NSM	1	1
Number of Class	NOC	16	18
Number of Interface	NOI		
Number of Packages	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the UML diagram. The code structure maintains a medium complexity, as several methods are created where various operations are implemented for evolution and validation of states. The code in general do not present errors, because it was eaten. The code came well documented, as it can identify the methods and attributes.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the use case diagram was generated 603 lines of code, of these lines were accounted 16 classes, 63 methods of which one is static, 24 attributes and 4 static attributes, 6 subclasses. Before proceeding to make a more detailed analysis of metrics, we can verify that the line number metrics is a reference metrics and to ensure that this number makes sense we have to analyze the other metrics because there is a dependency between them.

Starting with the first metric number of classes that exist in the code reflects the connections between the states of the diagram and how we can verify that there are 16 classes as well as there are 16 connections. By analyzing the number of methods, we can

verify in the code that for each state was created a method, it is found that for connection and for attributes was also created a method i.e. when we talk about methods, we can consider that each state is an attribute and for Each attribute was generated two methods. The attributes that were generated are consistent with the diagram we can see that through the code and the diagram, i.e. in the state diagram a state may have several sub-states for example in our case as we can observe the state "Self-Test" has several sub-state such as "Failure" that all sub-states will be reflected in attributes. Thus, we can conclude that the number of attributes that was generated corresponds to what was drawn in the diagram. There are 6 subclasses and they were generated because there are states that have sub-stated, checking the diagram we can conclude that. There are 4 statistical attributes that represent the initial and final states of the system. After qualitative and quantitative analysis, we conclude that the code generation was performed as expected.

Table 15 shows a reverse engineering use case summary table showing the results of code generation from the statechart diagram. The generated diagram consists of providing the complete implementation of the Bank ATM system using the Modelio framework. The generation of the diagram was made from the C ++ language and choosing this language for this specific diagram was explained earlier.

Table 15 – Results state machine diagram reverse engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	24	27
Number of Static Attributes	NSF	4	4
Number of Methods	NOM	63	68
Number of Static Methods	NSM	1	1
Number of Class	NOC	16	17
Number of Interface	NOI		
Number of Packages	NOP	1	1

Checking the diagram, we can see that the states of the statechart diagram that were generated and are code compliant. To validate the following we find that in the diagram are present the states and their composite states, which are well identified. Quantitatively we can see that 8 states were generated and a complex state and 3 static states that are the

initial state and the final state. Since validation cannot be done directly, as a state in the code is reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram.

Next, we must analyze the transitions between states. For this it was made as previously an interpretation of the code and analysis of the diagram and it was concluded that there are 1 transition that are with the changed directions, i.e. one the "Cancel" transition should leave the composite state and enter the "Idle state". "in the analysis of the code it was concluded that this error happened because there was another unnamed transition and the conflict arose changing the meanings. Quantitative 19 links were generated, looking at the diagram we can see that all the states are properly connected, which reflects what is in the code. We can conclude that the Modelio tool in general managed to generate the statechart diagram as intended, the existing inconsistencies were corrected. manually, but we can verify that tools do not deal well with transitions, to make mistakes of this genre we must have a code that follows the rules of good practice.

By analyzing the third case study, then is round-trip engineering we found that changes were made to the previously constructed statechart diagram. The changes that were made was to add a new state "Wait Transaction" within the composite state. It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged, but the new changes were added. Therefore, we can conclude that Modelio can easily handle changes to its long-life cycle diagram. After a change was made in the code, where it was added the same as was added in the diagram, respectively, updating the diagram it was found that the diagram was updated keeping the state unchanged, I add the state but the respective transaction that diverged to link to the remaining state. it was not added, but the sense of transitions within the composite state was changed. Obviously, it is difficult to understand in detail how Modelio determined the rules for generating these transitions, justifying it as an internal tool error. After analysis we conclude that applying round-trip engineering to the statechart diagram, updates the diagram successfully but with some inconsistencies, being critical to the diagram operation.

Table 16 shows a forward engineering use case summary table that shows the results of code generation from UML diagrams. The generated Java code consists of provides the complete implementation of the Online Shopping system activity diagram using the Modelio framework.

Table 16 – Results activity diagram forward engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	20	20
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	62	62
Number of Static Methods	NSM	0	0
Number of Class	NOC	12	12
Number of Interface	NOI		
Number of Packages	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the diagram. The generated Java code structure is constructed from the activity diagram, which maintains a medium level complexity of the generated code, bearing in mind that classes are created with methods where validation and control operations are implemented. The code in general do not present errors, because it was eaten. The code came well documented, because it is easy to identify the methods and what each attribute means.

From the activity diagram 542 lines of code were generated, from these lines were accounted 12 classes, 62 methods, 20 attributes and 6 subclasses. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics with us above to ensure that this number makes sense we have to analyze the other metrics because there is a dependency between them. Starting with the first metric number of classes that exist in the code relate the activities directly, i.e. the activity will give rise to one or more actions of an object, we can also say that a class here reflects the object, checking the diagram we can observe that there are 7 activities and assuming that in the current context each activity will reflect an object and also assuming that for object will have associated at least one class we can say that will be generated at least 7 classes, we can also observe that in the activity diagram there are Other elements such as starting points, decision points and assuming that the structure of a Java project, should be created a Service and Controller class where the logic will be done.

Moving on to the number of methods we can see that in the diagram there is activity that will give rise to action as we previously thought. But as each action will be reflected in the code, to answer this question I must realize that an action in the application will be associated with an attribute, and for attribute we can generate n number of methods. Looking at the current context of the diagram we observe that for each activity there will be more than one action, analyzing the code we can see that what is in the diagram is reflected in the code or we can conclude that there is consistency between the two.

Considering all the previous analysis and current context of the diagram we realize that the number of attributes is related to the actions, so the number of attributes generated is reflected in the diagram, so we conclude that the tool handles the attributes associated with a diagram well. of activity. Speaking of subclasses, which are 6, we can say that they were rotated in the context of having to take advantage of certain characteristics of another class, that is, there are classes that inherit characteristics of another, speaking of this we can verify that there are actions that have Similar characteristics As soon as the existence of subclass is normal, looking at the context beforehand, we can say that the number of subclass reflects what is in the diagram.

Table 17 show a reverse engineering use case summary table showing activity diagram that was generated from that of the Java code. The generated diagram consists of providing the complete implementation of the Online Shopping system using the Modelio framework.

Table 17 – Results activity diagram reverse engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	20	22
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	62	64
Number of Static Methods	NSM	0	0
Number of Class	NOC	12	12
Number of Interface	NOI		
Number of Packeges	NOP	1	1

Checking the diagram, we can see that the activities of the activity diagram that were generated and are code compliant. To validate the following we verify that in the diagram the activities that are well identified, as well as checking the pre-condition and post-condition duly marked at the beginning of each cycle of actions and at the end of them. Quantitatively we can see that 7 activities were generated. Since validation cannot be done directly because each activity is formed by various actions state in the code and reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram.

Next, we must analyze the transitions between the activities. For this as previously done, we will have to do an interpretation of the code and analysis of the diagram. After the analysis it was concluded that they are correctly generated, each activity is linked with the previous one, we also verify that there are moments of decision, and when checking the code, they are correctly constructed.

Given that these numbers cannot be directly accounted for in the code, we can state that 12 classes, 62 methods, 20 attributes and 6 sub-classes gave rise to 24 transitions and 9 decision moments, observing the diagram we can see that all the states are correctly connected. Qualitatively and qualitatively the Modelio tool did a good job, as I was able to generate the diagram completely without errors, so we can conclude that Modelio was able to generate the activity diagram as intended.

By analyzing the third case study, i.e. round-trip engineering, we found that changes were made to the previously constructed activity diagram. The changes that were made was to add a new activity "Remove from Shopping Chart". It was found that the code generated after this change reflects what is in the diagram i.e. the old code was unchanged, but the new changes were added. After making the change in the code, where it was added the same as was added in the diagram, respectively, updating the diagram it was found that the diagram was updated keeping the activities unchanged but adding the new activity but its link that diverged to link the state has not been added and removing the link between the "Add to Shopping Chart" activity. Being an internal error of the tool, it is difficult to understand in detail how Modelio has determined the rules to remove the following links. After analysis we conclude that applying round-trip engineering to the activity diagram, updates the diagram successfully but with some inconsistencies, which are critical to the operation of the diagram.

4.2.3. Applying Scenarios to Visual Paradigm

Table 18 shows a summary table of the forward engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of provides the complete implementation of the Hospital Management System class diagram using the Visual Paradigm-UML (VP-UML) framework.

Table 18 – Results class diagram forward engineering

Class Ciagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	12	14
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	27	28
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	56	54
Number of Static Methods	NSM	0	0
Number of Class	NOC	22	21
Number of Interface	NOI	0	0
Number of Interface	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated Java code skeleton is built from the UML class diagram, which keeps the complexity of the generated code low because no extraneous classes are created. The methods have a sturdy equivalent to the class diagram. The code in general do not present errors, because it was eaten. The code was documented, as it is possible to identify the attributes methods easily.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the class diagram was generated 236 lines of code, from these lines were accounted 21 classes, 54 methods, 27 attributes and 11 subclasses. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics is a reference metrics and to ensure that this number makes sense we have to analyze the other metrics because there is a dependence between them. By analyzing the number of classes in the diagram and the class number of the code we find that quantitatively they are equal so we can conclude that the classes were generated as expected. By moving to the following metrics, numbers of the methods and performing the same analysis that was done before, then is we can

conclude that quantitatively is correct because all the methods that were rotated derive from the attributes, to speak more specifically an attribute will generate two methods Set_atribute and Get_attribute Given that we have 27 attributes we can conclude that 57 methods were generated which is equivalent to number of attributes. Speaking of attributes as we saw 27 attributes were generated and, in the diagram, there were 27 so we can also conclude that the code generation went as expected. We can also state that 11 subclasses were generated then is a class that implements another class counts as a direct child of that class and analyzing the diagram we conclude that the number coincides. This analysis can also be verified in the graph below, where we can see that the total theoretical area covers the practical area in totality, being the theoretical total that is referred to the class diagram and the generated code practice.

Table 19 shows a reverse engineering use case summary table showing the results of generating the UML diagram from code. The class diagram consists of providing the complete implementation from the Hospital Management System Java code using the Visual Paradigm-UML (VP-UML) framework.

Table 19 – Results class diagram reverse engineering

Class Ciagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	12	14
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	27	28
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	56	54
Number of Static Methods	NSM	0	0
Number of Class	NOC	22	21
Number of Interface	NOI	0	0
Number of Interface	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can see that the classes were generated correctly, then is have the same type, the attributes were also generated as expected having the correct types of them, but in some cases having some attributes repeated. Associations in some classes were generated correctly and associations are generally present. Generalizations are also present and analyzed the

code, the previous diagram we can conclude that were generated as expected. Checking the diagram in general is an eligible diagram, structurally well accomplished.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 21 classes, 54 methods, 27 attributes, 4 associations and 13 aggregations. In making a more detailed comparison of the metrics between the generated diagram and the code, we find that the number of diagram classes and the class number of the code are quantitatively the same, but we also find that the descriptions and typology.

By moving to the following metrics, method numbers and performing the same analysis that was done before we can conclude that quantitatively is correct because all the methods that were rotated derive from the attributes then is the GETTERS and SETTERS where 54 methods were generated that reflects the number. Attributes which are 27. When checking the number of bindings that are the getters and the attribute setters that help in the binding of the classes, they are correct but as explained earlier they are exchanged. Verifying the same were the aggregations we can say that they at the quantitative level are equal. In this way we can conclude that VP-UML in general got the diagram that is structurally well done. This analysis can also be verified in the graph below, where we can see that the total theoretical area covers the practical area in totality, being the theoretical total that is referred to the class diagram and the generated code practice.

By analyzing the third case study, then is round-trip engineering we found that changes were made to the previously constructed class diagram. The changes that were made was to add a new class with 3 attributes and 2 methods and was linked to class "Person", this change was made to see the behavior of the tool when changing the structure of the diagram. It was found that the code generated after this change reflects what is in the diagram then is the old code was unchanged but one more class with its attributes and methods was added. After a change was made in the code, where it was added the same as was added in the diagram, respectively updating the diagram it was found that the diagram was updated with the new class keeping what was unchanged. Having this analysis, we conclude that for the class diagram, VP-IML can handle round-trip engineering.

Table 20 shows a summary table of the forward engineering use case showing the results of code generation from UML diagrams. The generated Java code consists of providing the complete implementation of the Sentinel HASP Licensing Components system diagram using the Visual Paradigm-UML (VP-UML) framework.

Table 20 – Results component diagram forward engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	5	5
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	14	14
Number of Static Methods	NSM	0	0
Number of Class	NOC	5	5
Number of Interface	NOI	9	9
Number of Packages	NOP	1	1

The case study helps us evaluate the code generated from the code generation tool. Qualitative analysis shows that the generated code is consistent with the UML diagram. The generated code structure corresponds to the diagram. The code in general do not present errors, because it was eaten. The code came well documented, since it is clear to identify the components, their artifacts as well as the interfaces and their methods.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the activity diagram 282 lines of code were generated, from these lines were accounted 5 classes, 12 methods, 9 interfaces and 1 package. In deepening this analysis to understand what these numbers tell us we must look at the component diagram and understand if they reflect what is in the code.

Before proceeding to make a more detailed comparison of metrics we can observe that the metrics of number of lines is not a metrics that can be observed directly in the diagram and to verify if this number makes sense, we must analyze the other metrics because there is a dependence between, they. Looking at the component diagram, we can see that the system has 9 components that provide and consume behaviors through interfaces. We can also affirm a component contains one or more artifacts, and each artifact can be composed of class, methods and attributes but in the context of our system

we can observe that the components contain information that cannot be accounted for only by making assumptions. Seeing the 5 classes that were generated and analyzing the diagram we conclude that they were generated with the structure of the Java project, where was created Service and Controller classes where will be made the logic and called the components that contain the artifacts. For this reason, we cannot make a direct relationship with the diagram. By checking the methods that were created we can say that they were generated due to the existing operations between components and interfaces. By checking the number of them we can say that they reflect what is in the diagram. The 9 interfaces that were created in the code serve to interconnect the components, and we can verify the same by looking at the diagram.

Table 21 how's a summary table of the reverse engineering use case that shows the results of UML diagram generation from code. The component diagram consists of providing the complete implementation from the Java code of the Sentinel HASP Licensing Components system using the Visual Paradigm-UML(VP-UML) framework.

Table 21 – Results component diagram reverse engineering

Component Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	0	0
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	9	5
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	18	14
Number of Static Methods	NSM	0	0
Number of Class	NOC	5	5
Number of Interface	NOI	12	9
Number of Packeges	NOP	1	1

The case study helps us evaluate the diagram generated from the Java code. Qualitative analysis shows that the generated diagram is consistent with the code. We can observe that the generated components are in conformity with the code, because it can be easily identified in the diagram, as well as their artifacts. We can also observe that the interfaces were generated correctly, then is each interface is correctly associated with each component, we can also observe that the

Associations between interfaces were generated as intended. Generalizations are also present and analyzing the code we can gather that were generated in the diagram as expected.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the Java code was generated 9 components, 10 interfaces, 16 associations and 2 aggregations. By analyzing in more detail the metrics between the diagram and the code, we observe that in the diagram there are 9 components and analyzing the code we identify the same number of components so we can say that the tool can identify the code in the component, we also observed that all the artifacts contained In the component were also undetected, the landfills that were used in the case are generic artifacts with no Papyrus.

Observing the diagram it is possible to verify that 10 interfaces were generated and knowing the context of the system in which the interfaces are used to communicate with the components we verify that each component has an interface that is quantitatively correct, there is a component that has two interfaces one that consumes information then is where the system receives information and other interfaces where it communicates with the other components. That said we can verify that the tool handles interfaces well in context of the component diagram.

The associations generated between the components quantitatively are correct. Verifying the same were the aggregations we can say that they at the quantitative level are equal to the diagram and correctly diffused. Thus, we can conclude that VP-UML got the diagram as intended, the inconsistencies that exist can have a big impact if not corrected, so it's a point to improve.

Table 22 how’s a forward engineering use case summary table showing the results of code generation from the statechart diagram. The code generated consists of providing the complete implementation of the Bank ATM system using the Visual Paradigm-UML framework (VP-UML), the choice of C ++ language for this case was because it has the same paradigm as C ++ ie it is a language oriented. objects, and you can do an analysis equivalent to what was being done in C ++. The reason for doing C ++ generation for the use case diagram was because Papyrus does not currently support Java for this diagram. The other tools that were chosen also support C ++, so it will be a valid comparison.

Table 22 – Results state machine diagram forward engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	5	7
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	26	24
Number of Static Attributes	NSF	4	6
Number of Methods	NOM	65	63
Number of Static Methods	NSM	1	1
Number of Class	NOC	22	18
Number of Interface	NOI		
Number of Packeges	NOP	1	1

Qualitative analysis shows that the generated code is consistent with the UML diagram. The code structure maintains a medium complexity, as several methods are created where various operations are implemented for evolution and validation of states. The code in general do not present errors, because it was eaten. The code came well documented, as it can identify the methods and attributes.

Now a quantitative analysis will be performed but always take into consideration the qualitative analysis. From the use case diagram 585 lines of code were generated, from these lines were accounted 16 classes, 58 methods of which one is static, 24 attributes and 4 static attributes, 6 subclasses. Starting with the first metric number of classes that exist in the code reflects the connections between the states of the diagram and how we can verify that there are 16 classes as well as there are 16 connections. By analyzing the number of methods, we can verify in the code that for each state was created a method, it is found that for connection and for attributes was also created a method this is when we talk about methods, we can consider that each state is an attribute and for Each attributes was generated two methods. The attributes that were generated are consistent with the diagram we can see that through the code and the diagram, this is in the state diagram a state may have several sub-states for example in our case as we can observe the state "Self-Test" has several sub-state such as "Failure" that all sub-states will be reflected in attributes. Thus, we can conclude that the number of attributes that was generated corresponds to what was drawn in the diagram. There are 6 subclasses and they were generated because there are states that have sub stated, checking the diagram we can conclude that. There are 4 statistical attributes that represent the initial and final states of

the system. After qualitative and quantitative analysis, we conclude that the code generation was performed as expected.

Table 23 shows a reverse engineering use case summary table showing the results of code generation from the statechart diagram. The generated diagram consists of providing the complete implementation of the Bank ATM system using the Visual Paradigm-UML (VP-UML) framework. The generation of the diagram was made from the C ++ language and choosing this language for this specific diagram was explained earlier.

Table 23 – Results state machine diagram reverse engineering

State Machine Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	5	7
Number of Overridden Methods	NORM	1	1
Number of Attributes	NOF	26	24
Number of Static Attributes	NSF	4	6
Number of Methods	NOM	65	63
Number of Static Methods	NSM	1	1
Number of Class	NOC	22	18
Number of Interface	NOI		
Number of Packages	NOP	1	1

Checking the diagram, we can see that the states of the statechart diagram that were generated and are code compliant. To validate the following we find that in the diagram are present the states and their composite states, which are well identified. Quantitatively we can see that 8 states were generated and a complex state and 3 static states that are the initial state and the final state. Since validation cannot be done directly, as a state in the code is reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram.

Next, we must analyze the transitions between states. For this it was done as previously an interpretation of the code and analysis of the diagram and the conclusion was reached that the same were generated well, which did not happen in the previous tools. This happened because the VP-UML has an advanced functionality that validates the code. before the diagram generation is done. Quantitative 19 transitions were generated, by looking at the diagram we can see that all statuses are properly connected,

which reflects what is in the code. We can conclude that the VP_UML tool in general managed to generate the statechart diagram as intended.

By analyzing the third case study, then is round-trip engineering we found that changes were made to the previously constructed statechart diagram. The changes that were made was to add a new state "Wait Transaction" within the composite state. It was found that the code generated after this change reflects what is in the diagram then is the old code was unchanged, but the new changes were added. After a change was made in the code, where it was added the same that was added in the diagram, respectively, updating the diagram it was verified that the diagram was updated keeping the state unchanged, adding the state with respective transaction that links to the other states. After analysis we concluded that by applying round-trip engineering to the statechart diagram, it successfully updates the diagram as well as correctly adds the new ones and their connections, as explained earlier VP-UML has a code analysis function which ensures a certain confidence what will be generated.

Table 24 shows a forward engineering use case summary table that shows the results of code generation from UML diagrams. The generated Java code consists of providing the complete implementation of the Online Shopping system activity diagram using the Visual Paradigm-UML (VP-UML) framework.

Table 24 – Results activity diagram forward engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	20	22
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	62	64
Number of Static Methods	NSM	0	0
Number of Class	NOC	12	12
Number of Interface	NOI		
Number of Packages	NOP	1	1

The generated Java code structure is constructed from the activity diagram, which maintains a medium level complexity of the generated code, considering that classes are created with methods where operations and validation are implemented. The code in

general do not present errors, because it was eaten. The code came well documented, because it is easily possible to identify the methods and each attribute.

From the activity diagram 534 lines of code were generated, from these lines 12 classes, 58 methods, 20 attributes and 6 sub-classes were counted. Before proceeding to make a more detailed comparison of the metrics between the diagram and the generated code, we can see that the line number metrics with us above to ensure that this number makes sense we have to analyze the other metrics because there is a dependency between them.

Starting with the first metric number of classes that exist in the code relate the activities directly, then is the activity will give rise to one or more actions of an object, we can also say that a class here reflects the object, checking the diagram we can observe that there are 7 activities and assuming that in the current context each activity will reflect an object and also assuming that for object will have associated at least one class we can say that will be generated at least 7 classes, we can also observe that in the activity diagram there are Other elements such as starting points, decision points and assuming that the structure of a Java project, should be created a Service and Controller class where the logic will be done.

Moving on to the number of methods we can see that in the diagram there is activity that will give rise to action as we previously thought. But as each action will be reflected in the code, to answer this question I must realize that an action in the application will be associated with an attribute, and for attribute we can generate n number of methods. Looking at the current context of the diagram we observe that for each activity there will be more than one action, analyzing the code we can see that what is in the diagram is reflected in the code or we can conclude that there is consistency between the two.

Considering all the previous analysis and current context of the diagram we realize that the number of attributes is related to the actions, so the number of attributes generated is reflected in the diagram, so we conclude that the tool handles the attributes associated with a diagram well. of activity. Speaking of subclasses, which are 6, we can say that they were rotated in the context of having to take advantage of certain characteristics of another class, that is, there are classes that inherit characteristics of another, speaking of this we can verify that there are actions that have Similar characteristics As soon as the

existence of subclass is normal, looking at the context beforehand, we can say that the number of subclass reflects what is in the diagram.

Table 25 show a reverse engineering use case summary table showing activity diagram that was generated from that of the Java code. The generated diagram consists of providing the complete implementation of the Online Shopping system using the Visual Paradigm-UML (VP-UML) framework.

Table 25 – Results activity diagram reverse engineering

Activity Diagram			
Metrics	Acronyms	Total Real	Total Theoretical
Number of Children	NSC	6	6
Number of Overridden Methods	NORM	0	0
Number of Attributes	NOF	20	22
Number of Static Attributes	NSF	0	0
Number of Methods	NOM	62	64
Number of Static Methods	NSM	0	0
Number of Class	NOC	12	12
Number of Interface	NOI		
Number of Packages	NOP	1	1

Checking the diagram, we can see that the activities of the activity diagram that were generated and are code compliant. To validate the following we verify that in the diagram the activities that are well identified, as well as checking the pre-condition and post-condition duly marked at the beginning of each cycle of actions and at the end of them. Quantitatively we can see that 7 activities were generated. Since validation cannot be done directly because each activity is formed by various actions state in the code and reflected through class, attribute, and methods but by interpreting the code we can see that they reflect what is in the diagram.

Next, we must analyze the transitions between the activities. For this as previously done, we will have to do an interpretation of the code and analysis of the diagram. After the analysis it was concluded that they are correctly generated, each activity is linked with the previous one, we also verify that there are moments of decision, and when checking the code, they are correctly constructed. Given that these numbers cannot be directly accounted for in the code, we can say that 12 classes, 58 methods, 20 attributes and 6 sub-

classes gave rise to 24 transitions and 9 decision moments, observing the diagram we can see that all the states are correctly connected.

Qualitatively and qualitatively the VP-UML tool did a good job as I was able to generate the diagram completely without errors, so we can conclude that VP-UML was able to generate the activity diagram as intended. By analyzing the third case study, then is Round Trip Engineering, we found that changes were made to the previously constructed activity diagram. The changes that were made was to add a new activity "Remove from Shopping Chart". It was found that the code generated after this change reflects what is in the diagram then is the old code was unchanged, but the new changes were added. After making the change to the code, where it was added the same as was added to the diagram, respectively, updating the diagram it was found that the diagram was updated keeping the activities unchanged but adding the new activity with its link linking state " Add to Shopping Chart ". After analysis we conclude that applying Round Trip Engineering to the activity diagram updates the diagram successfully.

Chapter 5 – Conclusions and Recommendations

Model round-trip engineering will be a key factor in many next-generation model-driven software development approaches as it will allow modelers to move freely between different system representations. We compared the Papyrus, Modelio, and Visual Paradigm tools in terms of round-trip engineering capabilities.

Manual comparison is needed to understand the interpretations and mappings used when generating the diagrams or code to understand if they are semantically correct or a qualitative comparison. With automatic comparison using metric data (Metrics, 2015), differences and similarities between models can be quickly and easily discovered, quantitative comparison. We clarify several issues that need to be addressed in round-trip engineering automation. First, we explain that there is a differenfce between round-trip engineering on the one hand and advanced and reverse engineering on the other.

After quantitative analysis of the metrics of the first scenario, Forward and Reverse Engineering without any changes, applied to the case studies, we can see that the tools all successfully followed the scenario. After the qualitative analysis we concluded that all the tools cope well applying in the first scenario, because they can semantically make interpretation, namely when the diagram was generated from the code, we saw in the analysis of the previous chapter that were generated all the classes and all the links between them. same, as well as semantically correct. We also concluded that by applying the first scenario, either for code generation or diagram generation, the tools always generate new diagrams or new code, with a new context.

Even applying the second scenario, Forward and Reverse Engineering with changes, under the same conditions as the first, the same happens, i.e. new code or diagrams with new context are generated, so there are no semantic errors. But when the third scenario, Round-Trip Engineering with changes is applied to all case studies, the story is different, because the changes are always applied in the same model and context, here we can observe in the analysis that semantic inconsistencies arose. Therefore, we can consider that round-trip engineering should be based on a clear definition of the consistency of the model used to locate possible inconsistencies. Once detected, round-trip engineering can use different model reconciliation strategies to make the model consistent.

In general, all the tools were able to successfully apply the scenarios and obtain positive results, but Visual Paradigm stood out for the quality of the generated code, as well as the quality of the diagrams. On the other hand, Papyrus stood out for its rich customization, for example it gives easy to define metamodels, which did not happen in the others. Modelio stood out in the ease of creating the models and generating the code. Another factor to note is that proprietary, i.e. paid applications offer faster support than open source applications which limits the resolution of various emerging issues.

Comparing the work of other authors, namely Khaled, we can see that there was a considerable evolution in the definition of the models, that is, it is done more formally, and the quality of the UML increases what makes the application of the round-trip engineering will be increasingly viable in software development. Flowing in inconsistencies, it was found that the inconsistencies that existed in the application of round-trip engineering that the author Wang had, some of them were solved with for example in the static classes now the precision increases when the code is generated, the attributes the respective types. are well identified and the links between the classes semantically well defined, the same can be verified in the analysis of the previous chapter, which, in turn, the generated code has better quality, not requiring large interventions.

Based on this discussion, we have proposed several qualities that we believe are desirable for Round-Trip Engineering approaches and suggest, in some cases, possible directions for solutions. The recommendations that the authors suggested were that the tools had to cover a larger consistency of the model, the same being done by applying MDA to metamodel, that is, creating increasingly generic model layers that can be applied to more models and models. contexts.

As part of future work, we are interested in investigating how we could integrate consistency definition tasks, decide consistency, and resolve consistency in a single, accessible approach. To do this we could automate the analysis of automatically generated diagrams using the SDMetric framework, which will make the semantic analysis more specific. This will allow room to evolve the tools based on metrics.

References

Vangel V. Ajanovski, *Round-Trip Engineering and Comparison of Open-Source and Free Tools for UML Modelling*, 14th Workshop on Software Engineering Education and Reverse Engineering, Sinaia, Romania, 2014.

Ardis, M., Daley, N., Hoffman, D., Siy, H. (2000) *Software Product Lines: A Case Study.* Software Practice and Experience, vol. 30, 7, 825-847. Ardis, M., Daley, N., Hoffman, D., Siy, H. and Weiss, D. (2000), Software product lines: a case study. Softw: Pract. Exper., 30: 825–847. DOI: https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<825::AID-SPE322>3.0.CO;2-1

Atkinson, C., Kühne, T. (2003) *Model-Driven Development: A Metamodeling Foundation*. IEEE Software, vol. 20, 5, 36-41. DOI: https://doi.org/10.1109/MS.2003.1231149

A Comparison between UML Tools. In Proceedings of the 2009 Second International Conference on Environmental and Computer Science (ICECS '09). IEEE Computer Society, Washington, DC, USA, 111-114. https://doi.org/10.1109/ICECS.2009.38

ModelGoon team (2011). *UML4Java: Bring UML Visual models into the Java World.* Available online at: http://www.modelgoon.org

Gene Wang, Brian McSkimming, Zachary Marzec, Josh Gardner, Adrienne Decker, and Carl Alphonce. (2007). *Green: a flexible UML class diagramming tool for Eclipse*. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). ACM, New York, NY, USA, 834-835. DOI: https://doi.org/10.1145/1297846.1297913

Alanen, M., Lundkvist, T., Porres, I. (2005) Comparison of Modeling Frameworks for Software Engineering. Nordic Journal of Computing, vol. 12, 4, 321- 342.

Alanen, M., Porres, I. (2008) A Metamodeling Language Supporting Subset and Union Properties. Software and Systems Modeling, vol. 7, 1, 103-124.

Ardis, M., Daley, N., Hoffman, D., Siy, H. (2000) Software Product Lines: A Case Study. Software Practice and Experience, vol. 30, 7, 825-847.

Atkinson, C., Kühne, T. (2005) A Generalized Notion of Platforms for Model-Driven Development. In Beydeda, S., Book, M. and Gruhn, V. (Eds.) Model-Driven Software Development. New York, Springer-Verlag, 119-136.

Atkinson, C., Kühne, T. (2003) Model-Driven Development: A Metamodeling Foundation. IEEE Software, vol. 20, 5, 36-41.

Brown, A. W., Conallen, J., Tropeano, D. (2005a) Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In Beydeda, S., Book, M.and Gruhn, V. (Eds.) Model-Driven Software Development. New York, SpringerVerlag, 1-16.

Brown, A. W., Conallen, J., Tropeano, D. (2005b) Practical Insights into Model-Driven Architecture: Lessons from the Design and Use of an MDA Toolkit. In Beydeda, S., Book, M. and Gruhn, V. (Eds.) Model-Driven Software Development. New York, Springer-Verlag, 403-431.

Brown, A. W., Iyengar, S., Johnston, S. (2006) A Rational Approach References 90 to Model-Driven Development. IBM Systems Journal, vol. 45, 3, 463-480.

Object Management Group. Available at http://www.omg.org.

Unified Modeling Language: Superstructure - version 2.1.1. Available at http://www.omg.org.

A. Kleppe, J. Warmer; “Do MDA Transformations Preserve Meaning? An investigation into preserving semantics”. MDA Workshop, York, UK, November 2003.

F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose; “Eclipse Modeling Framework”. Addison-Wesley Professional, 2003.

J. Kuester; “Consistency Management of Object-Oriented Behavioral Models”. PhD Thesis, University of Paderborn, March 2004.

G. Engels, J. Kuester, L. Groenewegen, R. Heckel; “A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models”. V. Gruhn (ed.): Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), ACM Press, Vienna, Austria, September 2001, pp.186-195.

J. Lind. Specifying Agent Interaction Protocols with Standard UML, In Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), LNCS 2222, Springer-Verlag, 2002

Booch, Grady & Rumbaugh, James & Jacobson, Ivar. (1999). Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). J. Database Manag, 2000

D. Batory, R. Cardone, Y. Smaragdakis, Object-Oriented Frameworks and Product Lines, Proceedings of the First Software Product Line Conference, 2000.

MDA specification found at http://www.omg.org/mda

S. Deelstra; M. Sinnema; J. Gurp, Model Driven Architecture as Approach to Manage Variability in Software Product Families. Workshop on Model Driven Architecture: Foundations and Applications June 26-27, 2003.

N. Fenton and S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, second ed. London: Int'l Thomson Computer Press, 1996.

S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476- 493, June 1994

G. Spanoudakis, A. Zisman. "Inconsistency Management in Software Engineering: Survey and Open Research Issues". Handbook of Software Engineering and Knowledge Engineering, (eds.) S.K. Chang, World Scientific Publishing Co., 2001.

Lano, K. and Haughton, H., The Z++ Manual. Technical Report, Imperial College, 1994.

France, R., Evans A., Lano K., and Rumpe B., The UML as a Formal Modeling Notation, Computer Standards and Interfaces, No. 19, pages 325-334, 1998.

Folwer M., UML Distilled, Addison-Wesley, 1997

France, R., J-M. Bruel, and M.M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modelling Environment, Journal of Object-Oriented Programming, 2000.

E. V. Sunitha and P. Samuel, ''Object oriented method to implement the hierarchical and concurrent states in UML state chart diagrams,'' in Software Engineering Research, Management and Applications. Springer, 2016.

E. Sekerinski, ‘‘Design verification with state invariants,’’ in UML 2 Semantics and Applications. Hoboken, NJ, USA: Wiley, 2009.

V. C. Pham, A. Radermacher, S. Gérard, and S. Li, ‘‘Complete code generation from UML state machine,’’ in Proc. 5th Int. Conf. Model-Driven Eng. Softw. Develop. (MODELSWARD), Porto, Portugal, Feb. 2017.

Bell, Donald, "UML basics: The component diagram", https://developer.ibm.com/articles/the-component-diagram, 2004.

Image was downloaded form https://www.uml-diagrams.org/examples/online-shopping-domain-uml-diagram-example.html

Image was downloaded form the book “Developing Applications with UML 2.2”

Study case was dounloaded from https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html?context=stm-examples

Study case was dounloaded from https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html?context=stm-examples

Metrics was dounloaded from http://metrics.sourceforge.net/

Desfray, P. (2010) ‘Using OMG Standards with TOGAF’, (October). doi: 10.1109/CEE-SECR.2010.5783155.

Hettel, T. (2010) ‘Model Round-Trip Engineering’, Computer. doi: 10.1081/E-ESE-120044648.

Lanusse, A. et al. (2009) ‘Papyrus UML: an open source toolset for MDA’, 5th ECMDA-FA: Proceedings of the Tools and Consultancy Track, pp. 1–4.

Visual and Paradigm, (2010) Visual Paradigm for UML 2. https://www.visual-paradigm.com/support/documents/vpuserguide/276/386_statemachine.html

Usman, M. and Nadeem, A. (2009) ‘Automatic Generation of Java Code from UML Diagrams using UJECTOR’, International Journal of Software Engineering and Its Applications, 3(2), pp. 21–38.

J. Rumbaugh, I. Jacobson, and G. Booch, Object-Oriented Analysis and Design with Applications, 3rd ed. Reading, MA, USA: Addison-Wesley, 2007.

Jouault, F., Bézivin, J. (2006) KM3: A DSL for Metamodel Specification. In Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 06). Bologna, Italy, June 14-16, Springer-Verlag.

Terrasse, M.-N., Savonnet, M., Becker, G. (2001) A UML-based Metamodeling Architecture for Database Design. In Proceedings of 2001 International Symposium on Database Engineering and Applications (IDEAS'01). Grenoble, France, July 16-18, IEEE Computer Society.

Brown, A. W., Iyengar, S., Johnston, S. (2006) A Rational Approach to Model-Driven Development. IBM Systems Journal, vol. 45, 3, 463-480.

Cook, S. (2004) Domain-Specific Modeling and Model Driven Architecture. Available at http://www.bptrends.com.

Sendall, S., Kozaczynski, W. (2003) Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software, vol. 20, 5, 42-45.

Hailpern, B., Tarr, P. (2006) Model-Driven Development: The Good, the Bad, and the Ugly. IBM Systems Journal, vol. 45, 3, 451-461.

Sendall, J. Kuester; “Towards Inconsistency Handling of Object-Oriented Behavioral Models”. Proceedings International Workshop on Graph Transformation (GT-VMT'04), Barcelona, Spain, March 2004.

A. Kleppe, J. Warmer; “Do MDA Transformations Preserve Meaning? An investigation into preserving semantics”. MDA Workshop, York, UK, November 2003.

France, R., J-M. Bruel, and M.M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modelling Environment, Journal of Object-Oriented Programming, To appear.

Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., Selic, B. (2004) An MDA Manifesto. Available at http://www.bptrends.com.

Booch, G., Jacobson, C., and Rumbaugh, J., The Unified Modeling Language - a reference manual, Addison Wesley, 1999.

Stoica, I., H-Abdel-Wahab, K. Jeffay, S. Baruah, J.E. Gehrke, and G. C. Plaxton: “A Proportional Share Resource Allocation Algorithm for Real-Time Timeshared Systems”, IEEE Real-Time Systems Symposium, Dec. 2000

J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual. Reading, MA, USA: Addison-Wesley, 1999

P. Clements & L. Northrop, Software Product Lines, Addison-Wesley 2001.

A. Cockburn. Writing Effective Use Cases, Addison-Wesley, 2001

Coffel, K. *et al.* (2010) *The national academies press*. doi: 10.17226/14402.