TECHNISCHE UNIVERSITÄT
CHEMNITZ

Design and Implementation of a Data Persistence Layer for the GEMMA Framework

Master Thesis

Submitted in fulfillment of the academic degree

M.Sc. in Automotive Software Engineering

Department of Computer Science

Chair of Computer Engineering

Submitted by:	Indhu Mathi Gowda
Student ID:	367442
Supervisors:	Prof. Dr. W. Hardt
	Philipp Helle (Airbus Group Innovations)

Abstract

Data within an organization is highly structured and organized into specified applications or systems. These systems have a different function within an organization, so each user will have a different level to access each system. So by the data mapping approach user can easily isolate those data and prepare the declarations for the available data element.

Generic Modular Mapping Framework (GEMMA) a new common generic framework for data mapping was developed by Airbus Group Innovation GmbH to avoid numerous potential issues in matching data from one source to another. It is geared towards the high flexibility in dealing with a large number of different challenges in handling huge data. It has an open architecture that allows the inclusion of the application-specific code and provides a generic rule-based mapping engine that allows the users to define their own mapping rules. But GEMMA tool is presently used to read and process the data on the fly in memory, as each time the tool is used for mapping data from different sources. This has an impact on large memory consumption when handling large data and is inefficient in storing and retrieving the session data which are the user decisions.

This paper provides a detailed description of the GEMMA tool, with the new concept for specific requirements inherited in the framework and in the current architecture to achieve the goals.

Acknowledgement

The successful completion of the Master thesis was possible only with great support of many people. I would like to take an opportunity to owe my gratitude to each and every one of them for their direct or indirect support.

I would first like to express my deepest gratitude to my supervisor Philipp Helle for giving me this wonderful opportunity to write my thesis at Integrated Product Engineering Team in Airbus Group Innovations. He has provided me continuous support and encouragement throughout the work of my thesis by his supervision and guidance. I would like to specially thank my second supervisor Wladimir Schamai. Both of my supervisors played a major role in the successful completion of my thesis by their advice and guidelines in gathering information required for research of this thesis. This helped me gain a huge knowledge in all the related topics of this thesis. They have been patient enough to clarify all my doubts and give me constructive feedbacks throughout the work and also extensive discussions about the complete architecture and present usage of the GEMMA framework.

I would like to express my sincere gratitude to Prof. Dr. rer. Nat. Wolfram Hardt, head of Automotive Software Engineering department at TU Chemnitz as this thesis work would have been incomplete without his guidance. I would also like to thank Stephan Blokzyl from TU Chemnitz for his prompt support and tutoring. By their technical advices, encouragement, attention to details and correction of my mistakes, technical guidance of this thesis work whenever required was most valuable for the successful development of the thesis.

Finally, I would like to thank all my family members and friends who have supported me throughout my thesis work with their unconditional love and encouragement.

List of Abbreviations

GEMMA	Generic Modular Mapping
HQL	Hibernate Query Language
SQL	Structural Query Language
ORM	Object Relational Mapping
JTA	Java Transaction API
API	Application Programming Interface
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
EJB	Enterprise JavaBeans
JBI	Java Business Integration
RCP	Rich Client Platform
GUI	Graphical User Interface
XML	Extensible Markup Language
OSGi	Open Service Gateway Initiative
JCS	Java Caching System
LD	Levenshtein Distance
JVM	Java Virtual Machine
LRU	Least Recently Used
RAF	Random Access File
GC	Garbage Collector
NMN	Number of Mappables and Nodes
OGMT	Old GEMMA Mapping Time
NGMT	New GEMMA Mapping Time
OGRT	Old GEMMA Resolving Time
NGRT	New GEMMA Resolving Time
OGMC	Old GEMMA Memory Consumption
NGMC	New GEMMA Memory Consumption

Table of Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

The biggest challenges encountered by most of the researchers today in relation of data mapping is: How to match the data from single or multiple data sources to data destination from similar or distinct data sources in a flexible and in an adequate way? [1] The Generic Modular Mapping Framework (GEMMA) gives an answer to this question by solving different kinds of mapping problems. This framework was designed to create an extensible and user-configurable tool that would allow a user to define the rules for mapping data. GEMMA processes data on the fly in the memory which leads to inefficiency of data persistency. Considering the drawbacks in GEMMA, this thesis presents solving inefficiency of the GEMMA tool.

1.1 Background

Data mapping is a process of creating a data mapping specification of data elements between two distinct data models based on specified rules. Figure 1 displays a data mapping application, in which data points from different data sources have relationships, also known as mappings. A mapping problem can now be defined as the challenge to identify mappings between data points from (potentially) different data points.

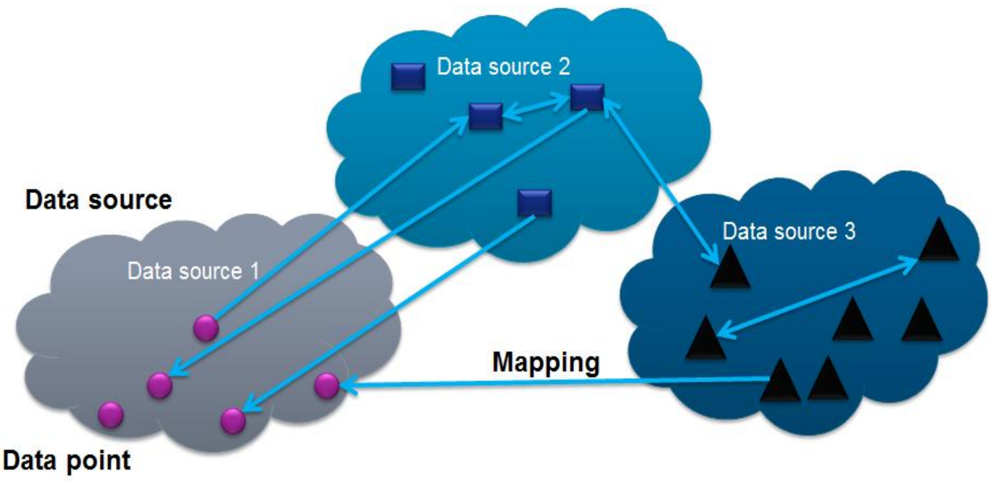


Figure 1 - Data Mapping

The Generic Modular Mapping Framework (GEMMA) is a framework which is designed to be a adaptable multi-purpose mapping tool for solving several kind of mapping problems that requires matching data points to each other with user defined distinct rules [1]. It is implemented in Java, open source libraries and frameworks are used. GEMMA is constructed upon the Eclipse Rich Client Platform (RCP), which prepare the strong basis for the GEMMA graphical user interface (GUI) and can work on every Windows PC except any other necessary installations. The Eclipse RCP enables the modular pluggable architecture of GEMMA and allows the distribution of GEMMA as an Eclipse product. GEMMA tool was developed to match huge data from multiple sources that is related in some way or the other, but not necessarily referencing the same object [1].

The GEMMA framework features an open architecture which creates an extensible and a user-configurable tool. The main matching process takes place in identification of relationships in data with the user defined rules, which produce a link between two distinct data models. It permits a user to define the rules for mapping data without any programming knowledge and yet still has the possibility to include an application-specific code to adapt to the needs of a concrete application. The GEMMA usage is generic for all kinds of application scenarios. A GEMMA project is defined by a GEMMA configuration which is stored in an XML file. This mainly defines the particular parsers, rules and exporters used in the particular rule-based mapping project.

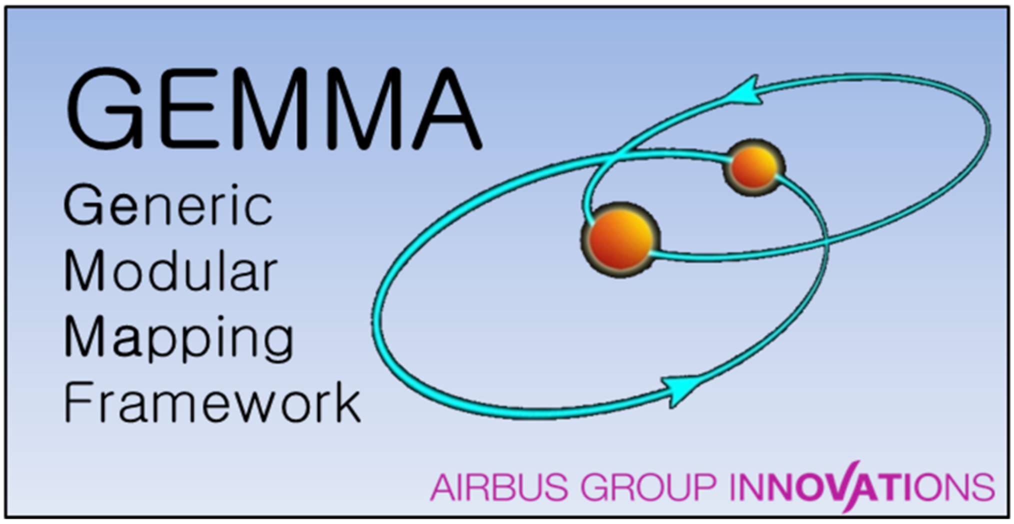


Figure 2 - GEMMA Display Image

1.1.1 GEMMA as Eclipse Plugins

GEMMA built on top of the Eclipse Rich Client Platform (RCP), which is a collection of frameworks that enables building modular, pluggable architectures. As Figure 3 depicts that the RCP provides some strong base services on top of which it is easily possible to build a custom application, that may consist of number of modules that works together in a flexible fashion.

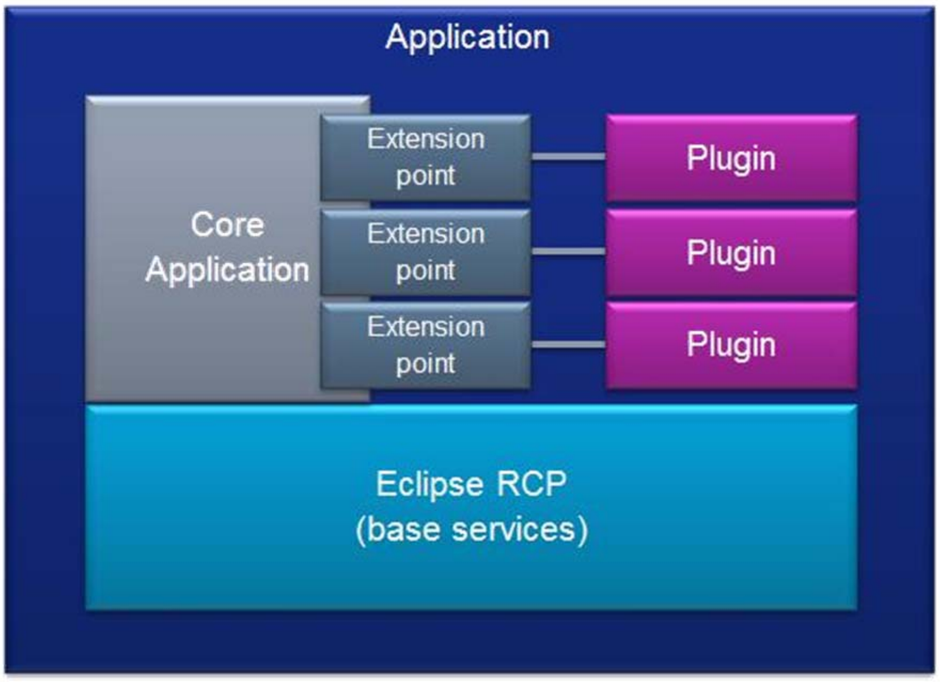


Figure 3 - Eclipse RCP

GEMMA is an Eclipse product and uses the Eclipse OSGi extension mechanism for registering and instantiating modules. This means that, as depicted by Figure 4, GEMMA is in essence a collection of Eclipse plugins, some of which can be selected by a user for specific applications, such as the data parsers or the exporters and some of which are fixed, such as the GUI. This architecture allows a tailored deployment of GEMMA. If some modules are not needed by a user or if a module must not be given to some users, it is possible to remove the plugin from the installation directory of GEMMA without the need for any programming.

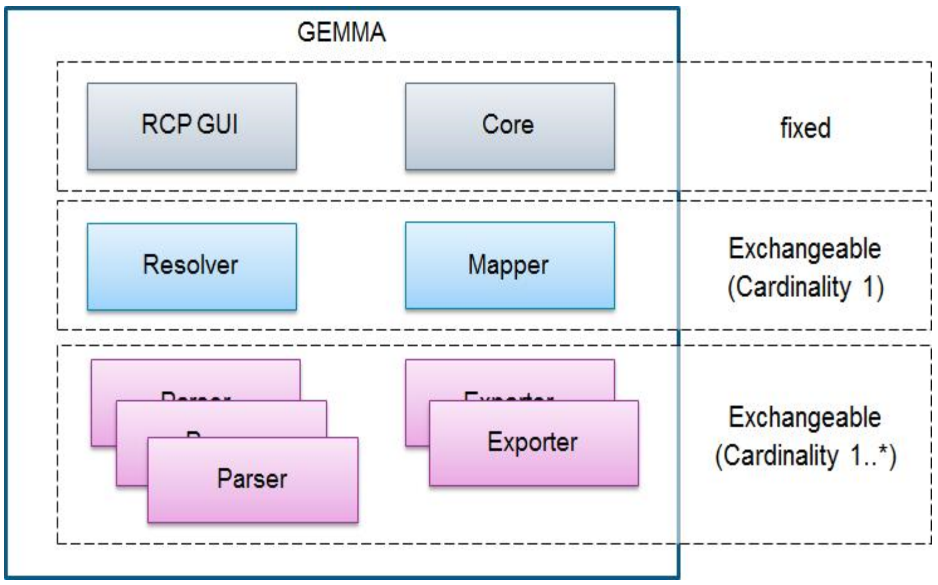


Figure 4 - GEMMA as a Collection of Eclipse Plugins

Only the plugins that are required by a mapping project configuration are needed and instantiated during the runtime as shown in figure 5.

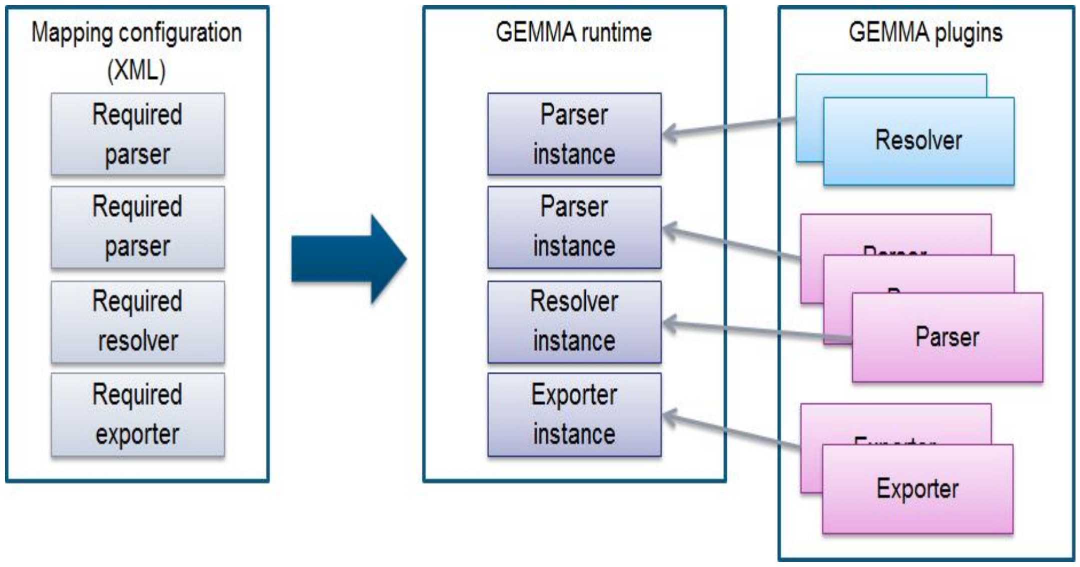


Figure 5 - Instantiation of GEMMA Modules at Runtime

1.2 Motivation and Objective

The main drawbacks of GEMMA motivate the work to increase its efficiency. The typical usage of the GEMMA tool is to map the two distinct data points from one or many data elements based on precise rules defined by the user. The complete data is stored in memory during runtime. This complete data stored in the memory will consume huge amount of memory during large data processing which increases memory footprints during runtime.

The other main drawback of the framework which limits the process is that the complete data is processed on the fly during the runtime and are not persistent. This means that a user cannot resume a session without re-computing the data every time the tool is started.

Considering these two main drawbacks which motivate the work with following two main objective of the thesis are:

- Reduce the memory footprint of GEMMA during runtime
- Enable storing and restoring the session data without re-computing the whole data every time

Apart from finding suitable solutions for the desired goals, the main approach is to identify the complications in the GEMMA architecture workflow which is leading for tool inefficiency.

1.3 Scope of the Thesis

Considering all the factors of GEMMA, the main aim of the thesis is to analyze the GEMMA framework and need for the data persistence in it; To establish a list of criteria based on the user requirements and other constraints that can be used to evaluate if a data persistence solution fits to the user needs.

There is a huge availability of options which may seem like a solution. The following high level task was considered during the project:

- Analysis of a GEMMA framework to implement new solution to satisfy the need for data persistency; Establish a list of user requirements and main constraints that can be tested against the exact fit to data persistency of user needs

- Compile a shortlist of solutions that could potentially be used to enable persistence in the GEMMA framework that are in line with the existing GEMMA workflow

- Solution evaluation of the identified solution against the established criteria and select the most beneficial solution

- Implement the finalized solution into the framework with the proper documentation

- Later develop evaluation criteria of the solution, test cases and procedures to verify and validate the solution

1.4 Structure of the Thesis

This thesis is structured as follows:

Chapter 1 gives an introduction to the GEMMA tool as an eclipse plugin, followed by the main motivation and objectives and the scope of the thesis work.

Chapter 2 gives an insight of how GEMMA actually works by its main concepts, rules and how the old GEMMA architecture was actually functioning with its applications. This chapter gives an in depth GEMMA functionalities, GEMMA graphical user interface and GEMMA installation process.

Chapter 3 presents the analysis of various solutions which would meet the requirements of thesis goals and the detailed description upon the selected approach. This chapter also discuss the GEMMA architectural limitations which was a barrier for the implementation of new approach and the solution to overcome this limitation.

Chapter 4 explains about the advancement in the GEMMA architecture as a solution for the old GEMMA architectural limitation, to meet the desired goal and the detail description of the selected approach with its prescribed features.

Chapter 5 gives an overview of the selected solution results. It describes the whole testing and validation section of the GEMMA tool along with the future work of the GEMMA application which can be performed.

Chapter 6 declares the final conclusion of the thesis work

Chapter 2

2 Introducing GEMMA

The Generic Modular Mapping Framework (GEMMA) is designed as a flexible multipurpose tool for any kind of problem that requires matching data points to each other. With the baseline of GEMMA framework in the introductory chapter, this chapter is the detailed description of the GEMMA for the clearest illustration of the tool. The following subsections will introduce the artifacts that make up the core idea behind GEMMA with suitable examples and describe the kind of mapping rules that can be implemented. Without these basic concepts of GEMMA framework, there cannot be any analysis of the GEMMA architecture in upcoming chapters.

2.1 Generic Modular Mapping Framework

The following subsections will introduce the requirements that were considered during the GEMMA development, the artifacts that makeup the core idea behind GEMMA, describe the kind of mapping rules that were implemented and show the generic process for the usage of GEMMA.

A. Requirements

The basic concept of a mapping tool is mapping of data that do not necessarily match completely in name, type, multiplicity or other details from different data sources to each other as depicted by Figure 1.

Relations between data from different sources and possibly in different formats are created. It is possible to output the generated relations in a user-defined format. This leads to a first draft of mapping tool, as depicted by Figure 6.

Data from different sources in different formats are read by mapping tool, then a mapping engine would create relations between the data and then the relations will be exported in different formats.

This is the minimum functionality a mapping tool must provide. In addition to this, there are three major requirements regarding the characteristics of the mapping tool, being **generic** in order to enable applications in different areas with similar challenges i.e. being **extensible**, as well as enable **user interaction**.

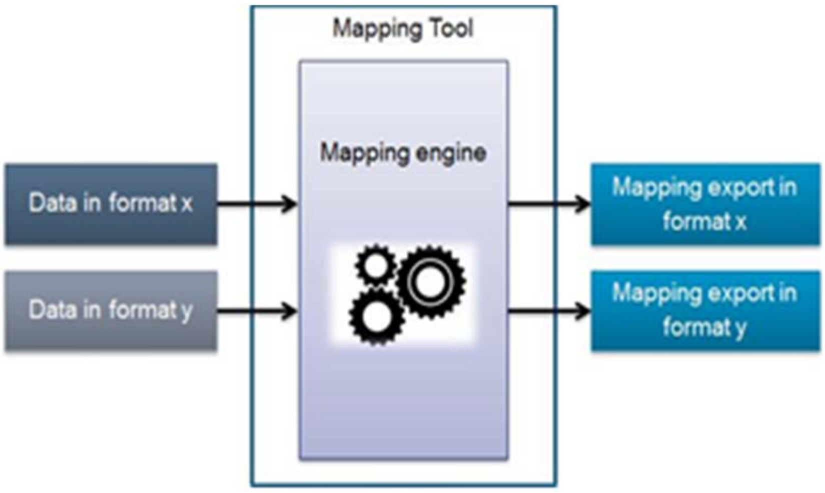


Figure 6 - Mapping Tool

B. Generic

The requirement for a generic tool stems from the fact that different mapping problems and challenges require different data sources and mapping rules.

This means that it shall be possible for the user to define the rules which govern the creation of mappings. The generic tool must read and interpret such rules in order to be able to create mappings between input data sets.

Additionally, it shall be possible to setup the current configuration of the mapping tool by means of user-defined configuration. Such project configuration contains information, such as where the input data is located, what mapping rules should be used and where the mapping export data should be written to. This discussion leads to an extension of second draft of the mapping tool which is shown in Figure 7.

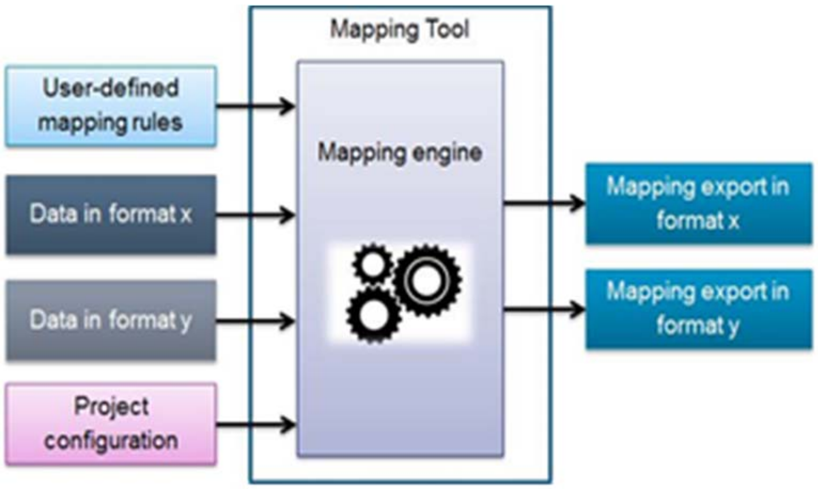


Figure 7 - Generic Mapping Tool

C. Modular

The requirement for modular software is an extension for the requirement that the software needs to be generic. Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains necessary information for executing only one aspect of the desired functionality if required.

GEMMA mapping tool anticipate very different contexts and applications with diverse data formats for import and export. To support this application, the GEMMA architecture needs to be modular by nature.

Standardizing the interfaces for importer and exporter modules allows creating new modules for specific applications without affecting the rest of the tool. Which perticualar modules are used in a specific mapping project can then be defined by the project configuration.

D. Interactive

Based on the assumption that the data in different data sources to be mapped can differ quite substantially in name, type, multiplicity or other details, it is resonable to assume that a perfect mapping is not always possible. This directly leads to the requirement that the mapping tool needs to be interactive. i.e. allow user-involvement when needed.

An interactive tool displays information to the user and allows user to modify the displayed data. Hence the GEMMA mapping tool shall be able to display the generated mappings between the input data and allow the user to modify these mappings using a Graphical User Interface (GUI).

In oder to present the generated mapping data to the user in a meaningful way it is necessary to condsider interpretation of the generated mapping data, This can be achieved by an additional module, the resolver module, which is an application-specific module. It is aware of application-specific module. It is aware of application-specific requirements and features. Using this information the resolver can process the genrated mapping data and provide information regarding validity of the generated mapping data to the user.

2.1.1 GEMMA Core Concepts

GEMMA is centered on a set of core concepts that are depicted by Figure 8. In an attempt to increment the flexibility of GEMMA tool functionalities, the core concepts have been defined in an abstract fashion. These concepts explain the unique functionality and its role in the rule based mapping tool. Each artifact’s functionality is explained in detail, with real - world examples below.

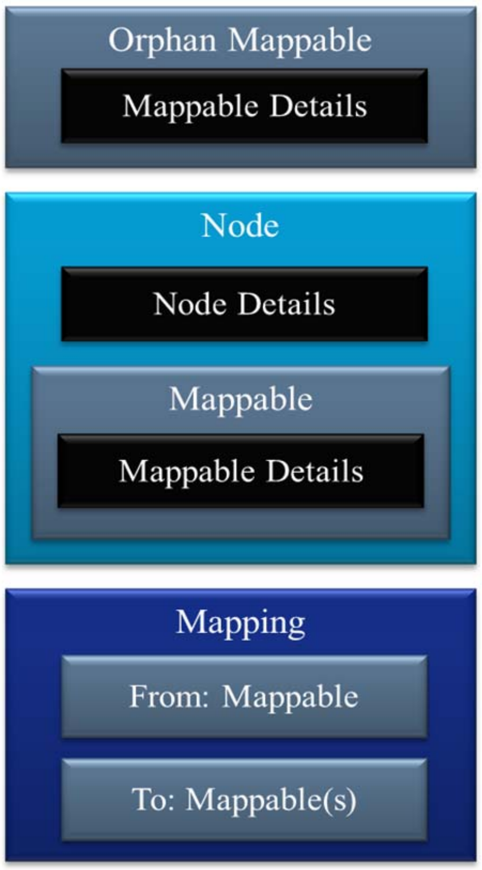


Figure 8 - Overview of Relevant Artifacts (from [1])

As shown in Figure 8, GEMMA makes use of the following core concepts:

- **Node –** A Node concept in GEMMA is, a group of one or more mappables

- **Mappable –** A mappable concept in GEMMA is, anything that has a property that can be mapped to other property corresponding to the particular mapping rules or user defined rules
  Orphan mappables are the mappables whose dominating node is not known or not appropriate to the complication specified in particular [1]. GEMMA will only generate data mappings for mappables that belong to a perticular node

- **Mapping –** The outcome of the specific application of mapping rules, i.e., a relation between one FROM mappable and one or more TO mappables. In general mapping is a connection established between the mappables [1]
  Note that the semantic interpretation of a data mapping extremely dependant on the application scenario [1]

- **Mapping rule** – Mapping rule is a function which determines how mappings are generated, i.e. how a single mappable can be associated to other mappables [1]. In general, it is a criterion that defines the mapping between mappables

- **Mappable or node detail –** This detail is a supplementary attribute of a mappable or a node in the form of a {detail name: detail value} pair. Mappable or node details is a choice which is not compulsory rule and can be defined in the context of a specific application scenario. There can be one or more details for a single mappable or node.

To emphasize these abstract definitions, Figure 9 present a simple example where real world objects depicted on the left hand side of the figure are represented in the form of GEMMA concepts on the right hand side of the diagram.

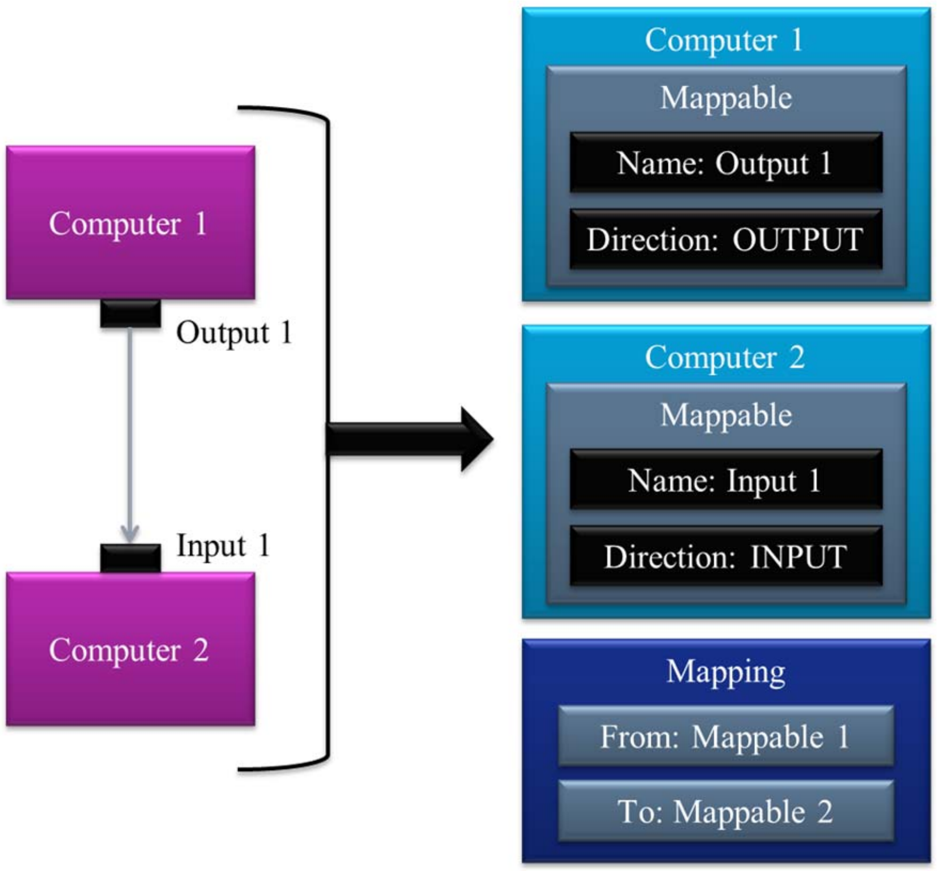


Figure 9 - Basic GEMMA Example (from [1])

In this example the GEMMA concepts are used as follows:

- **Node –** A computer with input and output signals
- **Mappable –** An IO signal of a computer
- **Mapping rule –** Output signals must be connected to input signal according to some criteria (e.g.: Same signal name or same data type etc.)
- **Mapping –** The connection established between Output 1 and Input 1

Another simple example to illustrate the core concepts of GEMMA is depicted in Figure 10. This gives an exact idea and a clear image of all the GEMMA core concepts. In this example, as previously explained that a node or mappable can have more than one detail according to its specific application scenario, there are two details for each mappable. And the basic mapping rule for this particular example is that to map both customer and city with an equal Zip Code value. The detailed description of a node, mappable, mapping and mapping rules are explained below with specific details. This is the most basic rule that the GEMMA can use.

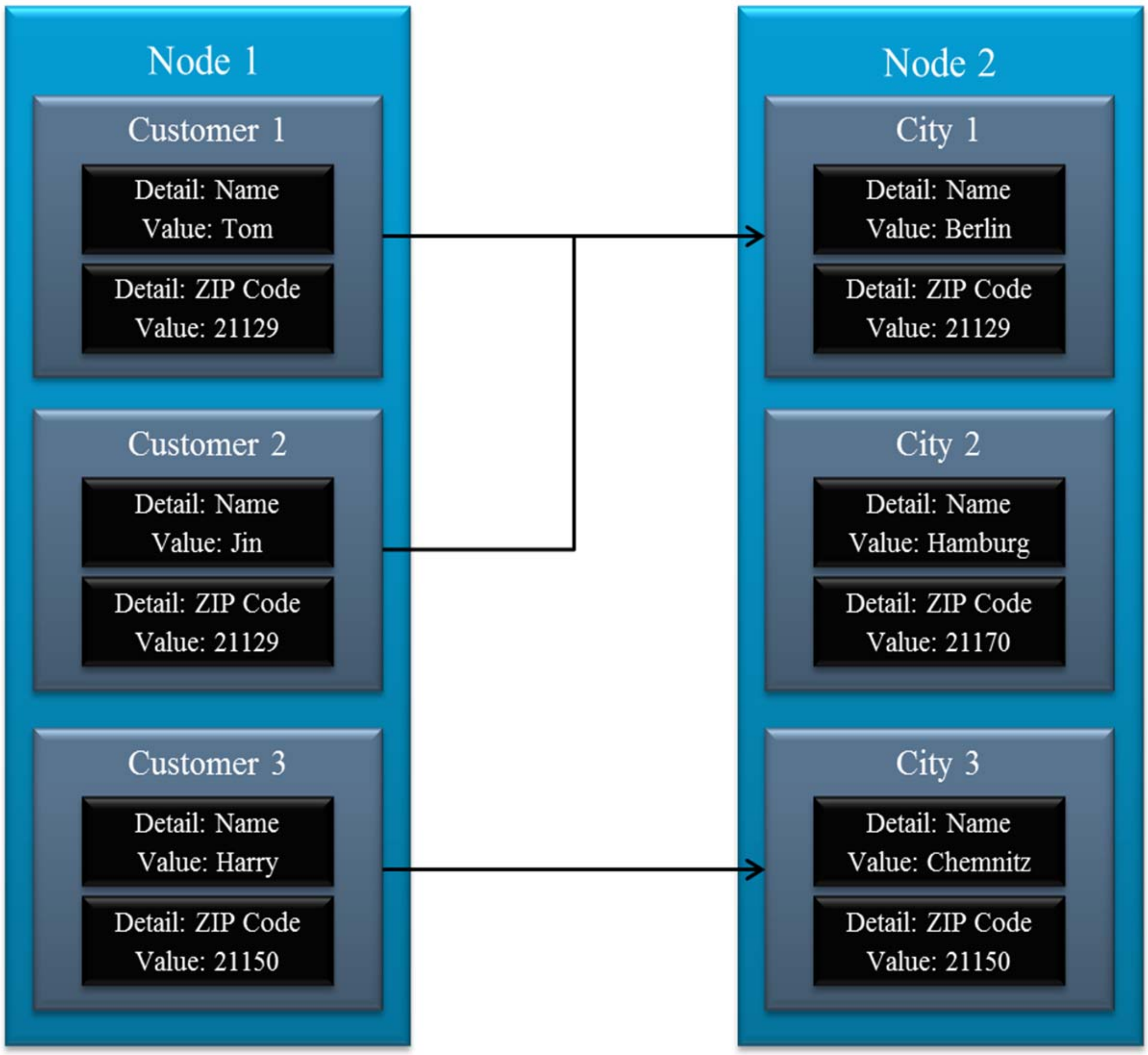


Figure 10 - Simple Mapping Example

Here in this example the GEMMA concepts are used as follows:

- Node – Node 1 and Node 2 with collection of mappables and its details
- Mappable – The node 1 and 2 contains three mappables of Customer and City respectively. It has the detailed Name and Zip Code value respectively
- Mapping rule – Map a customer to a city that has the same zip code as the customer

- Mapping – Connection established between Customer 1, 2 and City 1 and also Customer 3 and City 3 where it satisfies the mapping rule i.e. same Zip Code value

2.1.2 Mapping rules

A mapping rule is a function that specifies how mappings are created, i.e. how one mappable can be mapped to other mappables. Currently, GEMMA supports the following types of mapping rules that can be combined to define how each mappable can be mapped to other mappables. The mappings can be of any kind, i.e. one-to-one, one-to-many, many-to-many mappings.

- **Exact matching:** E.g., map a mappable to other mappables with the exact same name

- **Fuzzy matching or Approximate string matching:** E.g., Map a mappable to other mappables with an identical name (similarity can be based on the Levenshtein distance (LD), i.e., “put” can be matched to “pot” if we allow an LD of 1) [1]

- **Wildcard matching:** E.g., map a mappable to mappables that contain a certain value

- **RegEX matching:** E.g., map a mappable to mappables based on a regular expression

- **Tokenized matching:** E.g., split a mappable property into tokens and then map to another mappable with a property that contains each of these tokens in any order

- **Details:** E.g., match a mappable with value of detail X=x or more specifically, map a mappable with particular detail direction=” output” to mappables with specific detail direction=” input” [1]

- Structured rewriting of a searchable term is based on name, details and additional data, e.g., constructing of a new string based on the properties of a mappable and some given string elements and make a name matching with a new string (e.g., new string = “ABCD::” + $mappable.detail(DIRECTION) + “:: TBD::” + $mappable.detail(LOCATION) would influence to a search for other mappables with the name “ABCD:: Input:: TBD:: Front”) [1]

- **Semantic annotations** is a rule in which a user-predefined possible mappings (binding) are utilized as mediators, e.g., The recorded name of a mappable as a client of a mediator is matched to all mappables whose name is recorded as a provider of the same mediator [1]

So, by any series of the above mentioned type of rules can be utilized by the user to define mapping rule. For example, structured rewriting can be enforced on the target mappables, which would in effect mean determining aliases for each mappable in the mappable database in the context of a user rule [1]. In one GEMMA rule set, several rules can be defined for the same mappable with options for defining their application and prioritize, e.g. only if the rule with highest priority does not find any matches then rules with a lower priority are evaluated.

These mapping rules for a GEMMA project can be stored in one or more GEMMA rule files which have to conform to the XML schema GEMMARules.xsd.

2.1.3 GEMMA Process

The process for the usage of GEMMA framework is generic for all kinds of application scenarios where it consists of five steps as shown in Figure 11:

- Import
- Pre-Processing
- Matching
- Post-Processing
- Export

The mapping process is configured using XML configuration file that defines which parsers, rules, resolvers and exporters will be used in the mapping project. The open aspect of GEMMA framework allows implementing different data parsers for importing data, resolvers for post-processing of the mappings data and data exporters for exporting data [1].

Import loads data into the GEMMA framework. GEMMA framework provides the interfaces DataParser, MappableSource and NodeSource to define a recent data parser for an application-specific configuration of GEMMA. Later the data will be stored in the mappable database.

Pre-Processing of data involves selection of mappables that requires matching using described mapping rules. The rules used in a particular mapping project will be defined by the configuration.

Matching process comprises of running queries on the mappable database to find suitable matches for each selected mappable for mapping [1].

Post-Processing or match resolving is an alternative level that is highly compelled with specific application. It requires the interaction with the user to make a selection based on requirements. Post-processing also permits the user to implement the graphical user interface (GUI) to review and validate the produced mapping results, and thereby to check the completeness and correctness of the user defined rules and to manipulate mappings manually if needed [1].

Export is highly application specific which transforms the internal data model into an application specific output file.

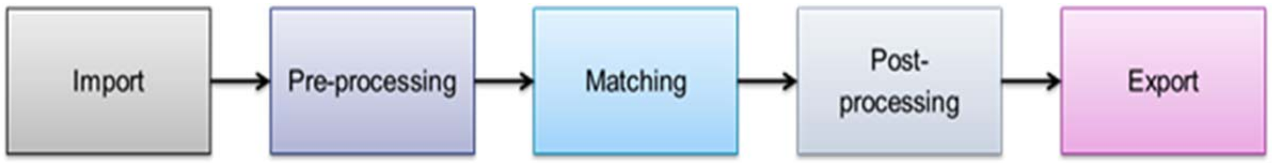


Figure 11 - GEMMA Generic Process

2.2 GEMMA Architecture

GEMMA (Generic Modular Mapping) architecture is depicted in the Figure 12.

GEMMA modules can be categorized either as core or as application specific modules. The core components are common to all GEMMA usage scenarios, whereas the application-specific components have to be developed to implement features that are very specific to achieve a certain goal. For example, data parsers are application-specific as applications might need data from different sources, whereas the mappable database and query engine is a core component that is shared.

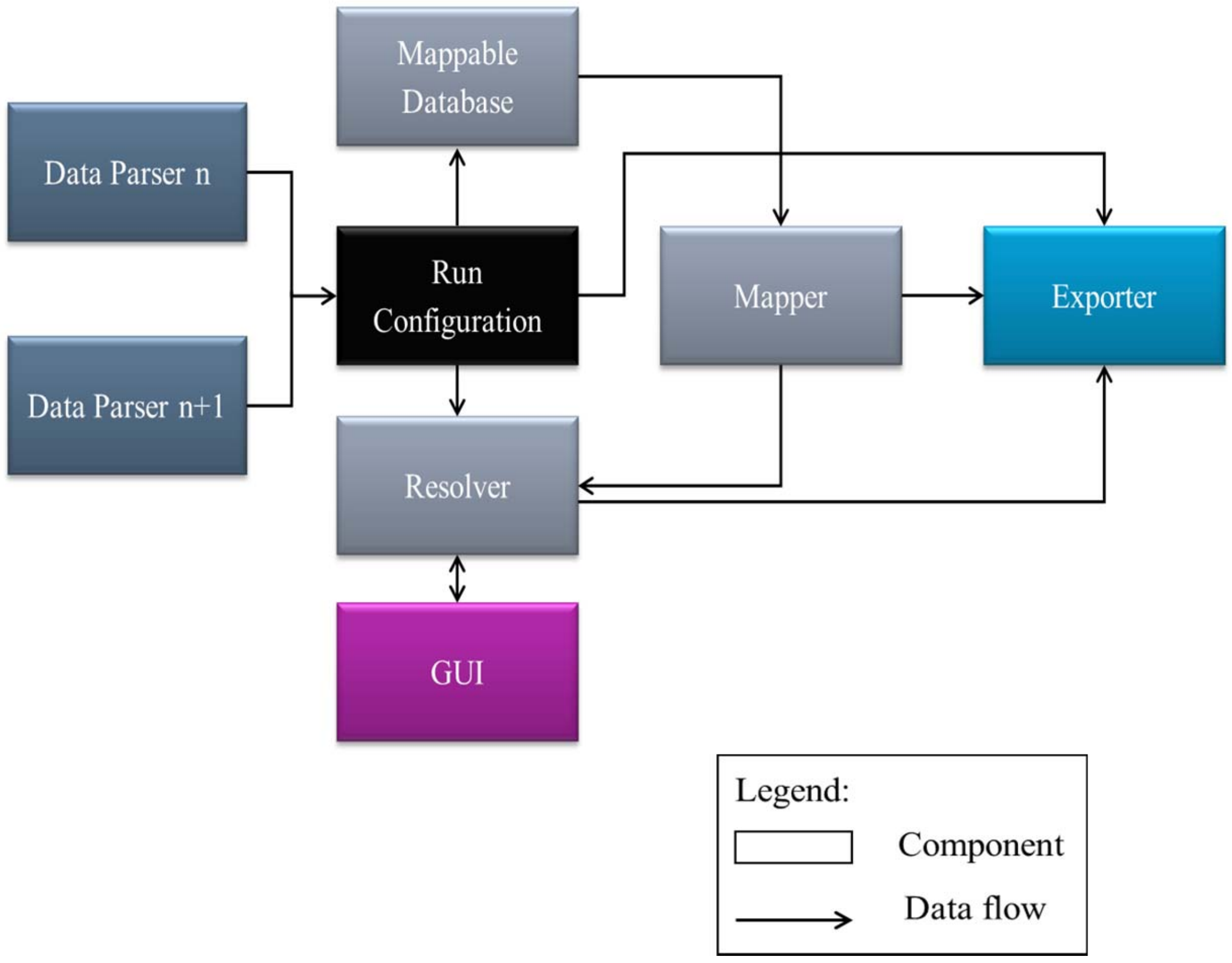


Figure 12 - Generic Mapping Framework Architecture

GEMMA architecture in Figure 12 denotes the representation of the five process steps in the application scenarios. The orderly data flow in the GEMMA architecture is explained as follows:

- One or many application-specific **Data Parser** read data, converts it into the form of nodes and/or mappables and import it into the GEMMA framework. All the available parsers are registered in an internal parser registry.

- **Run Configuration** will instantiate, configure and run those parsers, which are required by the configuration file in it. Run Configuration as a core component holds the configuration that defines which parsers, exporters and rules are used in the current mapping project.

- Later the data from the Run Configuration is sent to store in the **Mappable database.** This mappable database uses a full-text search engine called Lucene. So all the relevant data information is converted into strings. Each mappable is assigned a unique ID from its parser and other required information is stored as a detail-value pair in so-called fields as shown by Figure 13 [1].

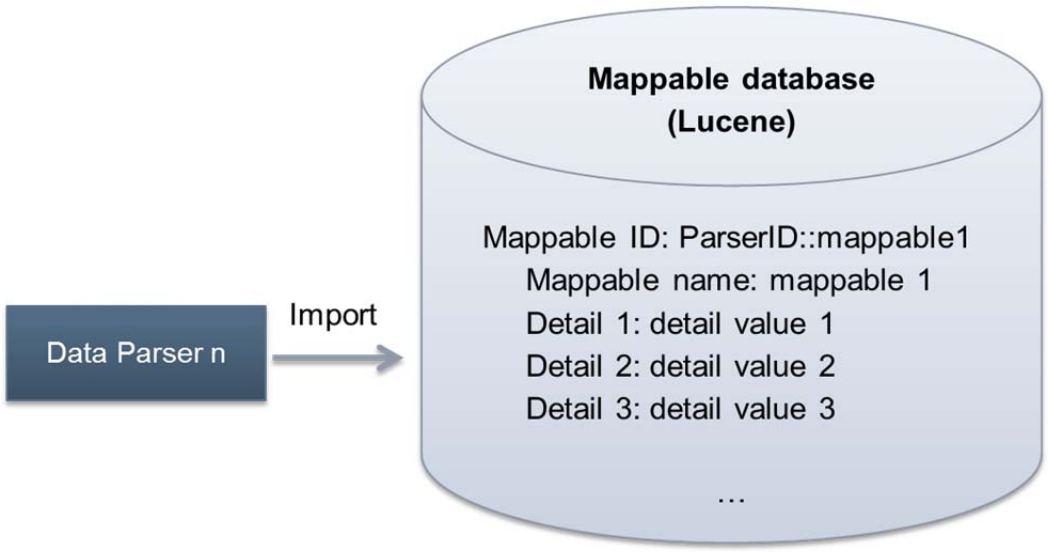


Figure 13 - Import Process

- Matching involves running queries on Lucene Database to find suitable matches for each mappable that is selected for mapping. The queries are derived from mapping rules. A mapping is a one to many relations between the mappables and all the matches that were found.

- As depicted by Figure 14, during the matching process, the mapper requests the mappables that exits in all parsers through the run configuration. For every mappable the mapper requests the application rules from the rule manager and uses the information on these application rules to generate a query that be used to query the mappable database. The results of the query are then used to create a mapping. These mappings are stored in **Mapper**

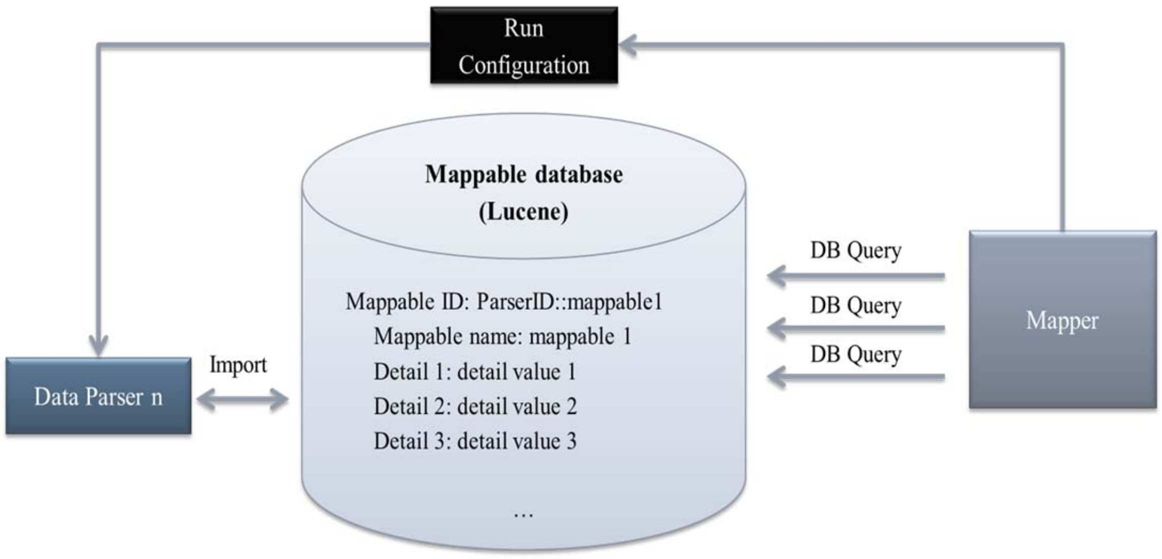


Figure 14 - Matching Process

Then the mappings are sent to **Resolver** which is an optional step, where it resolves mappings based on application specific semantics. During this process it allows user to apply graphical user interface to review and validate or to select the generated mapping results.

- An Exporter mainly involves transformation of the internal data model into an application specific output file, e.g. it can be just an XML file as the standard exporter produces, but it can also be an exporter directly into an application using the application's API. How the data are exported is completely encapsulated in the exporter.

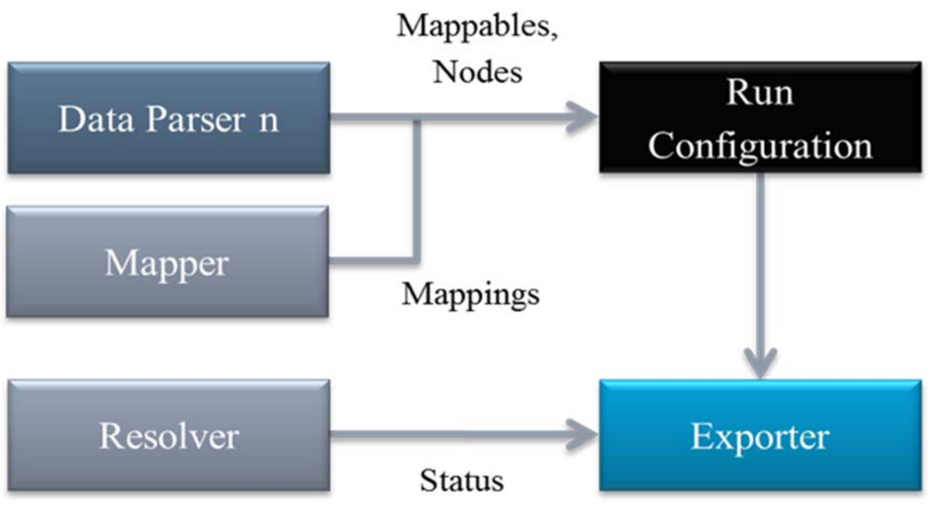


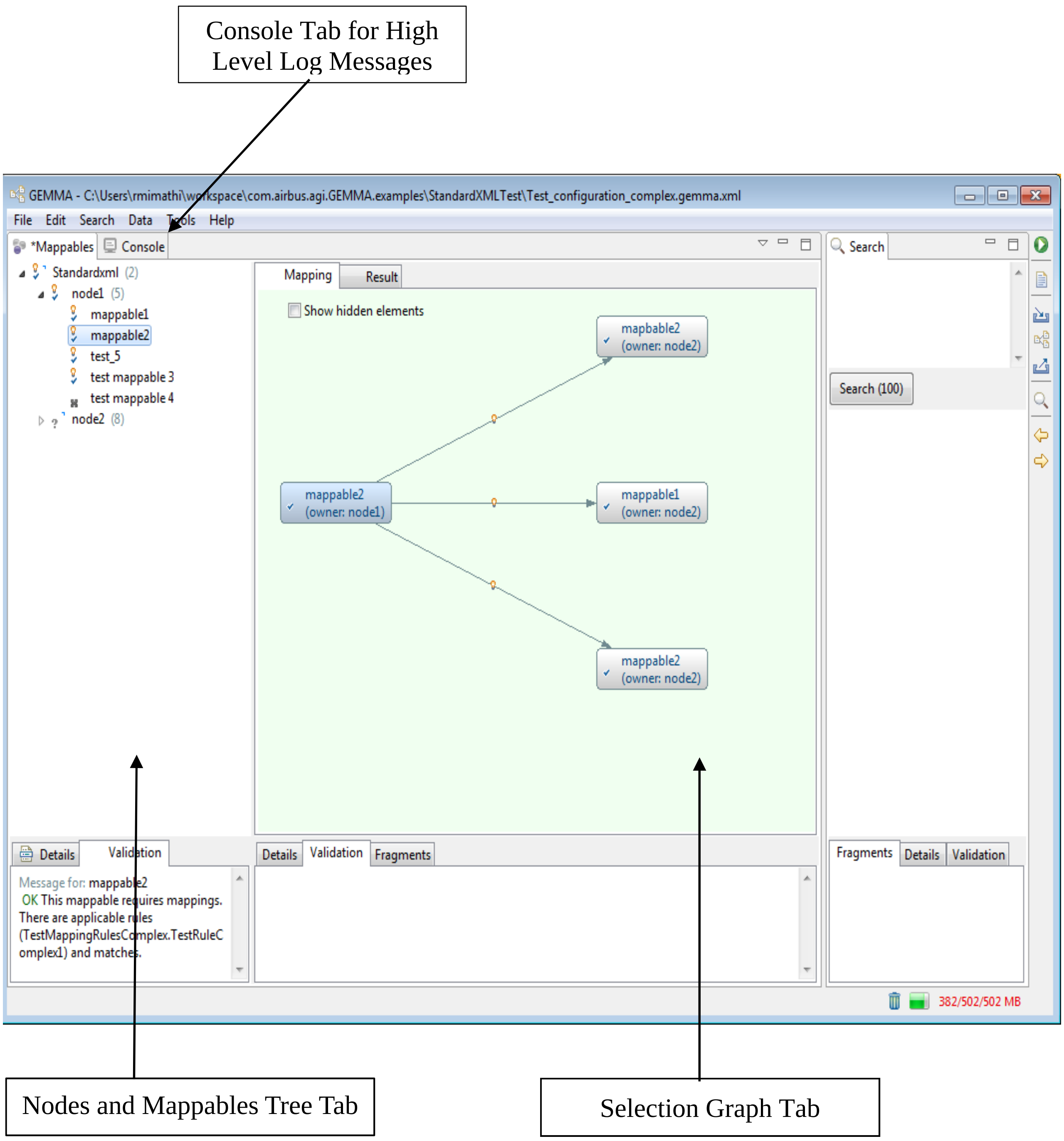
Figure 15 - Export Process

Each Exporter can obtain the available mappables, nodes and mappings through run configuration from data parsers and mapper respectively, and the status of the elements by the resolver as interpreted by Figure 15. Using this information the exporter creates a mapping export.

2.2.1 GEMMA GUI

GEMMA is built upon the Eclipse Rich Client Platform (RCP) [1], which provides the basis for the GEMMA GUI. It enables GEMMA's modular pluggable architecture and allows the distribution of GEMMA as an Eclipse product.

The GEMMA GUI contains the main operations like import data for mapping, export mapped data, load configuration file, etc. on the right side as shown by Figure 16. The Nodes and Mappables Tree displays nodes and mappables used in that particular mapping project, which have been imported by the Data Parsers. The Selection Graph section displays the automatically created mapping results for the selected mappable from the tree. The lower parts of both sides in Figure 16 are used to display the details or validation messages for a selected mappable or node.

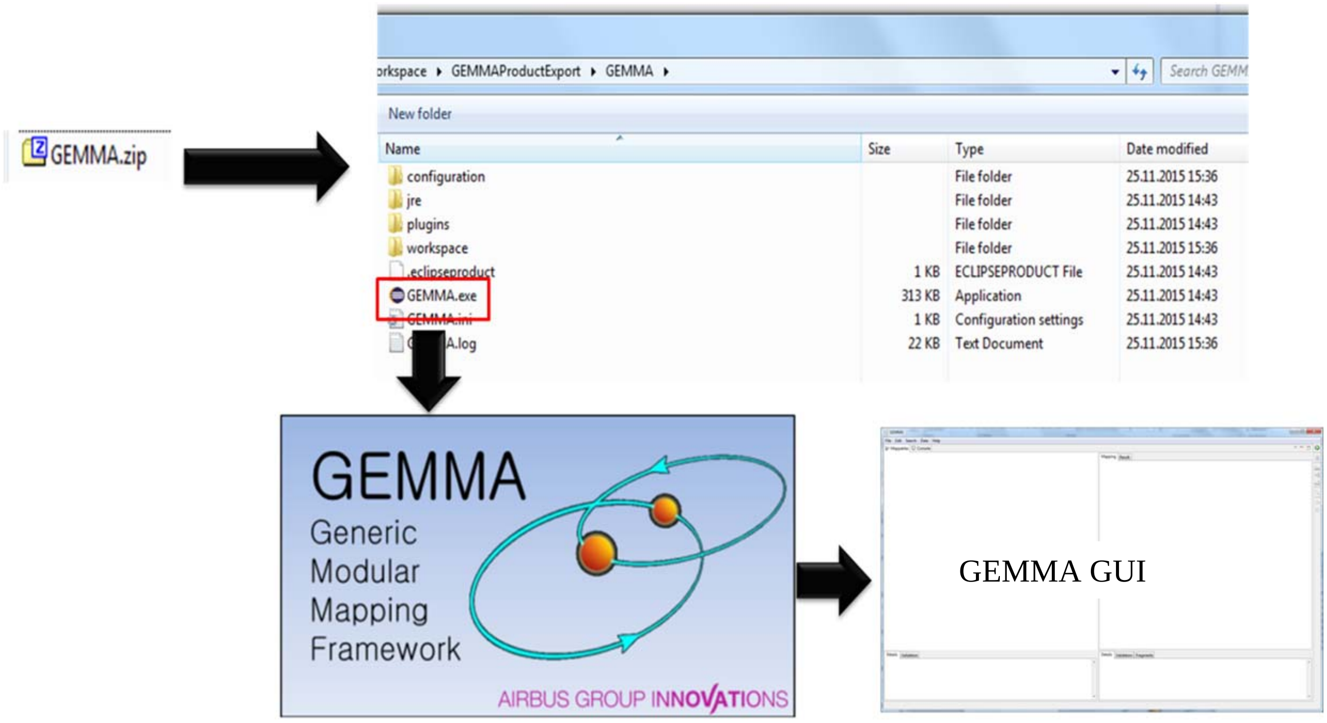


Message for: mappable2
OK This mappable requires mappings.
There are applicable rules
(TestMappingRulesComplex.TestRuleComplex1) and matches.

Figure 16 - GEMMA Editor

2.2.2 Running GEMMA

GEMMA can run on every 64-bit Windows PC without any other installations. GEMMA is distributed as an Eclipse product in the form of a .zip file which must be unzipped into a working directory. GEMMA is started by double-clicking the GEMMA.exe file in the GEMMA working directory as shown in Figure 17. This will start the loading process, display the GEMMA splash screen and then finally open the GEMMA main GUI.



Name	Size	Type	Date modified
configuration		File folder	25.11.2015 15:36
jre		File folder	25.11.2015 14:43
plugins		File folder	25.11.2015 14:43
workspace		File folder	25.11.2015 15:36
.eclipseproduct	1 KB	ECLIPSEPRODUCT File	25.11.2015 14:43
GEMMA.exe	313 KB	Application	25.11.2015 14:43
GEMMA.ini	1 KB	Configuration settings	25.11.2015 14:43
G A.log	22 KB	Text Document	25.11.2015 15:36

Figure 17 - GEMMA Initialization Process

2.2.3 GEMMA Application

Here is a simple application of simulation model composition from the public aerospace use case of the CRYSTAL [17] project. It consists of component models such as flight scenario profile, ice accretion dynamics, and tables for temperature of liquid water content. All of the component models must be interconnected. For example, the temperature profile component requires the current aircraft altitude, which is provided by the flight scenario component; the ice accretions dynamic component requires the current aircraft speed, which is also provided by the scenario component, etc. The individual models were built using the Modelica tool Dymola and exported as Functional Mockup Units [18] (FMUs) in order to be integrated, i.e. instantiated and connected, in a co-simulation environment [1].

There are more than 20 components which require high manual effort to connect. When more than 20 components without any connections is produced to GEMMA with the desired rule set, GEMMA resolves the model with automatic connections which are more than 50 connections as shown in Figure 18.

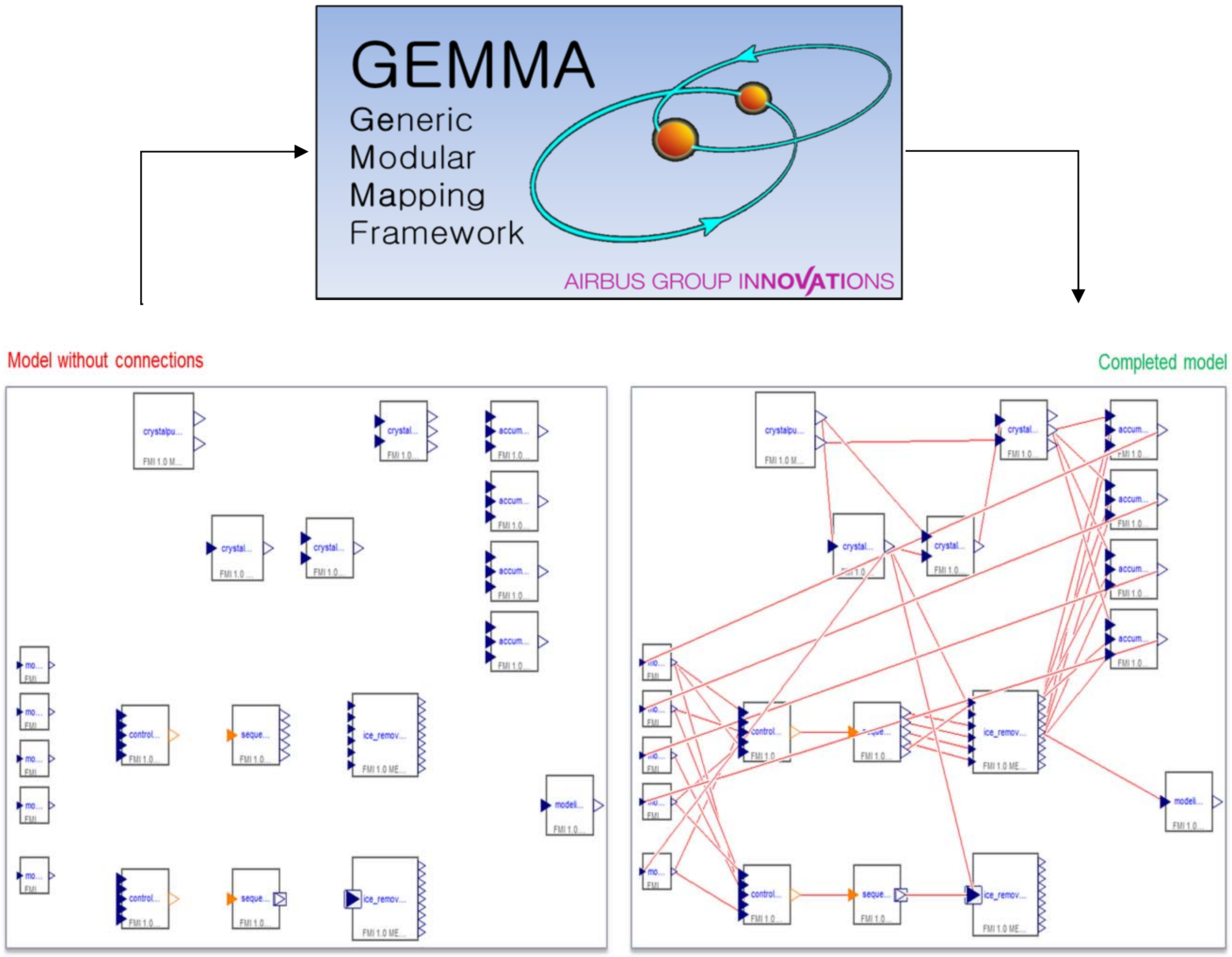


Figure 18 - Simulation Model Composition

Chapter 3

3 Analyze the Need for Data Persistence in GEMMA

This chapter will describe the topics researched during the thesis work. The available proprietary solutions and approaches used in the current market for solving two main goals of GEMMA are discussed. Various interfaces and frameworks have been investigated to evaluate the solutions for establishing criteria. Different methods were seemed to be a solution in the beginning, but were not appropriate for the particular project in some situations, so the appropriate solution is planned to be chosen for implementation.

3.1 Related Work

The main task was to understand GEMMA framework and the need for data persistence in it: to establish a list of criteria based on user requirements and other constraints that can be used to evaluate a data persistence solution that fits to the user needs. Few shortlisted solutions which initially pretended to be the solution to enable the data persistence in GEMMA framework was tested and evaluated based on references.

Serialization – Serialization is the transformation of an object into a stream of bytes in order to save the objects or transmit it to memory, database or a file. The main principle of serialization is to save the states of an object in order to be able to recreate or retrieve data through the deserialization process whenever needed by the user as shown in Figure 19. Java programmers can use the default Java serialization mechanism or can use their own custom Serialization methods [12].

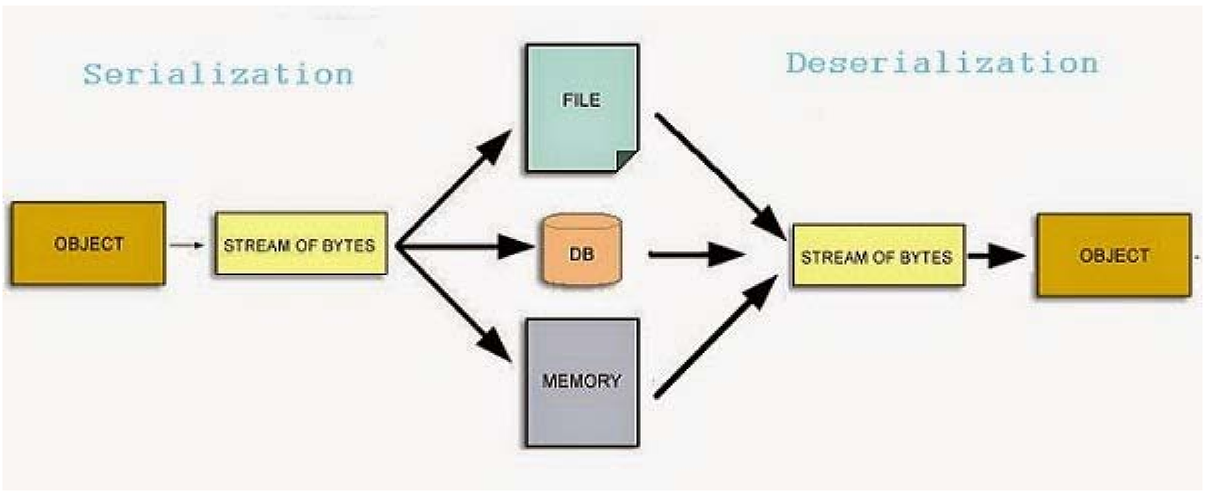


Figure 19 - Serialization Mechanism (from [3])

There are different serialization techniques available in the market with both advantages and disadvantages for serialization in Java as the GEMMA framework is completely developed in a Java environment. Some of them are presented and tested for the particular project requirements and the best serialization for Java is selected in which solution was developed.

Built-in Serialization – Java has a built-in serialization process where the entire process is JVM independent. When a Java class implements the java.io.Serializable interface, the JVM will take care of serializing objects in default format. Classes that do not implement this interface will not have any of their state serialized or deserialized [2].

Built-in serialization writes down the fully qualified class name at the beginning of every serialized instance, or else the search during deserialization will be unsuccessful, by this there will be an increase of serialized object file with a lot of type information which will be verified when the object is deserialized. So it conserves arbitrary object graphs (all the other serializers flatten graphs to trees), so to do this Java built-in serializer keeps track of every object's status which is an expensive operation [5] with huge memory consumption.

Kryo Serialization – Kryo Serialization is a fast and efficient object graph serialization framework for Java. Kryo serialization first registers the classes to serialize and then the objects can be written and read. No interfaces, mapping files or any other actions beyond registration are needed to serialize the objects using kryo. The serializers are implemented by default to read and write the huge data in various ways. If these implementations do not meet particular needs, they can be replaced in part or as in whole. The definition of distinct approach which flows from objects to bytes and bytes to objects are produced by the abstract class of Serializer. These Serializer has two approaches that can be enforced. Writes the object as bytes from the write() function. Creates a new instance of the objects and reads from the input to populate it from the read() function [4].

When kryo serializer writes down an instance of an object, initially it may demand to write down something that determine the object's class. Automatically, the fully qualified class name is written, and then the bytes for the object are written. Consecutive view of object type within the similar object graph are written utilizing a variable length int. Writing the full class name is considerably inefficient, so classes can be registered earlier [4]:

```
Kryo kryo = new Kryo();
    kryo.register(SomeClass.class);
    // ...
    Output output = ...
    SomeClass someObject = ...
    kryo.writeObject(output, someObject);
```

Here SomeClass in the example code is registered with kryo, which accomplice the class with an int ID. While kryo serializer writes away an instance of SomeClass, it will write away the int ID. This is more efficient than a Java built-in serialization, which writes out the whole class name at each instance. During kryo deserialization, the registered classes must have the perfect identical IDs they had during serialization. The registered approach as depicted in above example code commits the following available, lowest integer ID, which means that the order classes which are registered are essential. The ID can also be stated accurately to make order insignificant [4]:

```
Kryo kryo = new Kryo();
    kryo.register(SomeClass.class, 0);
    kryo.register(AnotherClass.class, 1);
    kryo.register(YetAnotherClass.class, 2);
```

The IDs are written most intensively when they are small, positive integers [4]. This ID method in kryo serialization reduce the size of the object file and make it performance efficient. A Kryo serailizer instance, can be utilized to automatically read and write the objects in a few distinct manners. For Kryo to automatically serialize the objects, it must know what serializer it must use. Classes that should be serialized can be registered with Kryo instance and a serializer can be specified.

Built-in Serialization vs Kryo Serialization

Both built-in serialization and kryo serialization methods were selected for the comparison and to choose the best out of it with the main criteria to be satisfied with the project. The main characteristics chosen for comparison were mainly the performance in terms of Serialization and deserialization process speed, size of the serialized file, memory consumed by the serialized file placed on the ram and the impact on the existing code. Table 4.1 shows the comparison characteristics and the compared characters below.

Characteristics	Serialization options for Java	
	Built-in	Kryo
Performance/ Speed	-	+
Size	-	+
Memory Consumption	-	+
Impact on existing code	-	+

Table 1 - Serialization Test Platform

As seen in Table 1, the Kryo serialization process wins against the built-in Java serialization process in all the characteristics selected for comparison according to the test reference in [5]. So the selected solution for session storage is Kryo Serialization process

After finding a solution for session storage, the next challenge is to find an appropriate approach for reducing memory footprint during runtime. A detailed analysis must be organized to test the correct approach by the best impact on the existing project code and few are discussed because of time constraint. Initially Kryo serialization was examined to serialize the data and store it on the disc, but it didn’t work out because when every time the deserialization starts all the serialized object must be loaded into the ram and then retrieved back, which is not an appropriate solution. Some of the few approaches which were examined for the desired project will be discussed below.

Hibernate – Hibernate is an Object-Relational Mapping (ORM) result for Java and it has been an open source persistence framework. Hibernate involves process of mapping Java classes to the database tables and from Java data types to the SQL data types. Hibernate acts in between the traditional Java objects and the database server to manage all the job in persisting those objects based on an appropriate object relational mechanisms and patterns as demonstrated in Figure 20. [6] The Java classes can be mapped to database tables by utilizing the configuration of an XML file or by utilizing the Java Annotations. Hibernate administer an SQL influenced language called Hibernate Query Language (HQL) which allows SQL-like queries to be written adjacent to Hibernate’s data objects. Hibernate is utilized together in standalone Java applications and Java EE applications utilizing servlets, EJB session beans and JBI service factor [7].

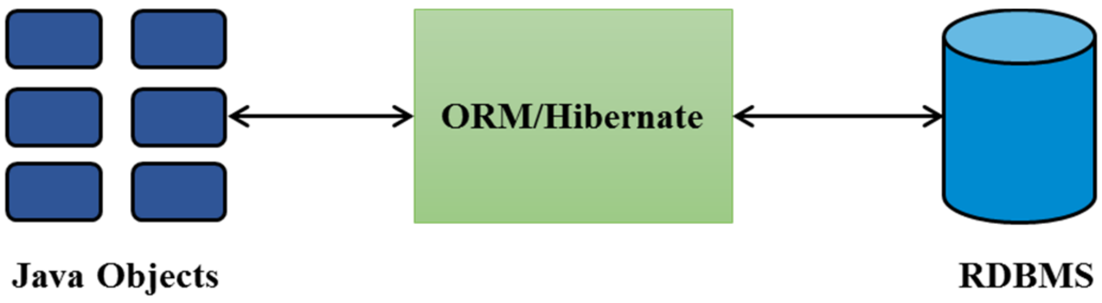


Figure 20 - Object Relational Mapping (from [6])

Hibernate utilize the database and configuration data to provide persistence services (and persistence objects) to the application. Figure 21 exhibit the complete view of the Hibernate Application Architecture with few important core classes. Hibernate uses various existing Java APIs like JDBC, Java Transaction API (JTA) and Java Naming and Directory Interface (JNDI). JDBC gives a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers [6].

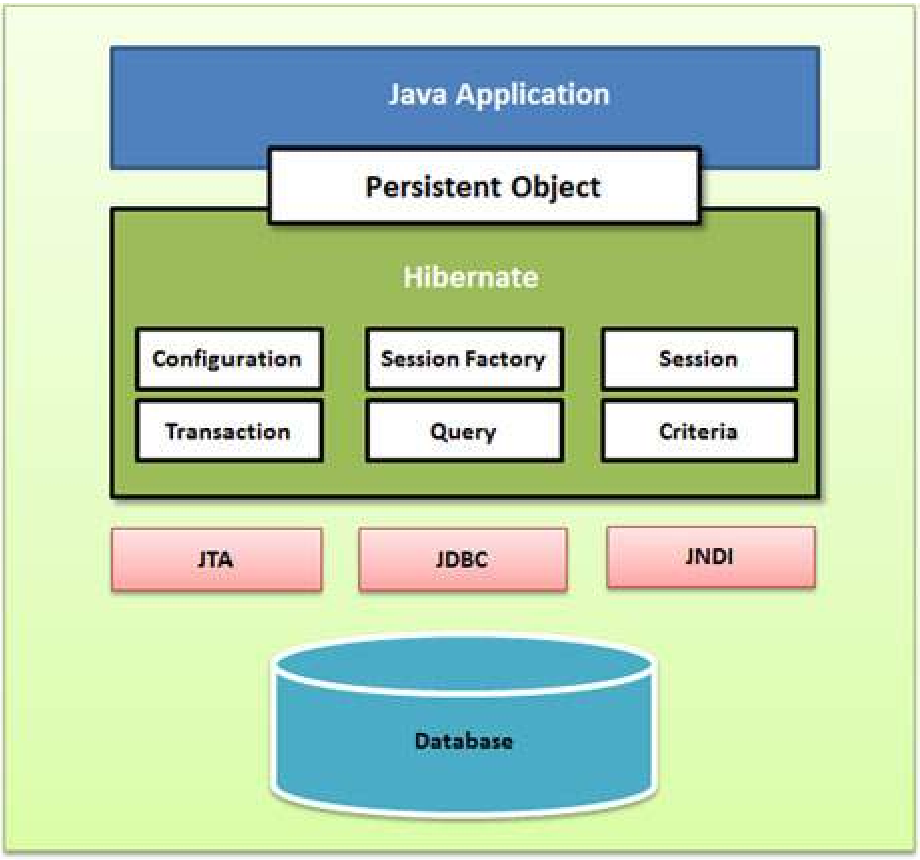


Figure 21 - Hibernate Architecture (from [6])

This Hibernate approach was tested by a sample example code. By this sample implementation and by the referenced results there are a few drawbacks identified which is not a good fit for the expected approach of GEMMA project. Hibernate has a performance cost as it adds a layer over JDBC.

- Performance requirements are not met to the expectation as it is very slow for the huge number of data
- Ease of implementation is bit high
- It generates lots of SQL statements during runtime which is extremely inefficient
- Generates complex queries with many joins during mapping from data-to-tables

When were there are changes to persistence tier you should check the SQL statements which are executed, which is highly impossible for huge data projects. And also it has a lot of table configuration information which is read at the beginning of the application which slows down the application process. Considering all these drawbacks of Hibernate it was not chosen to be the implementation solution for GEMMA.

Caching System

Caching systems are widely used across every IT industry. A cache is a temporary section to store various frequently accessed data so that the data can be retrieved or accessed in a very short period of time when compared the to normal repository. The idea is based on retrieving an object from this temporary memory should be significantly faster than retrieving the same object from its original source [24]. The main perception behind choosing caching system was that the frameworks have the data overflow to disk feature.

If the client invokes a request of desired data to the directory, it first checks the cache. When the entry can be established with a tag matching that of the desired data, then this data is used without having to access the main data storage. This is known as Cache Hit [8]. If the desired data not found in the cache, then this is known as Cache Miss at this stage a hit to the Back Storage is made to fetch the data back to the cache. Figure 22 demonstrates the whole process of Cache hit and Cache miss [8].

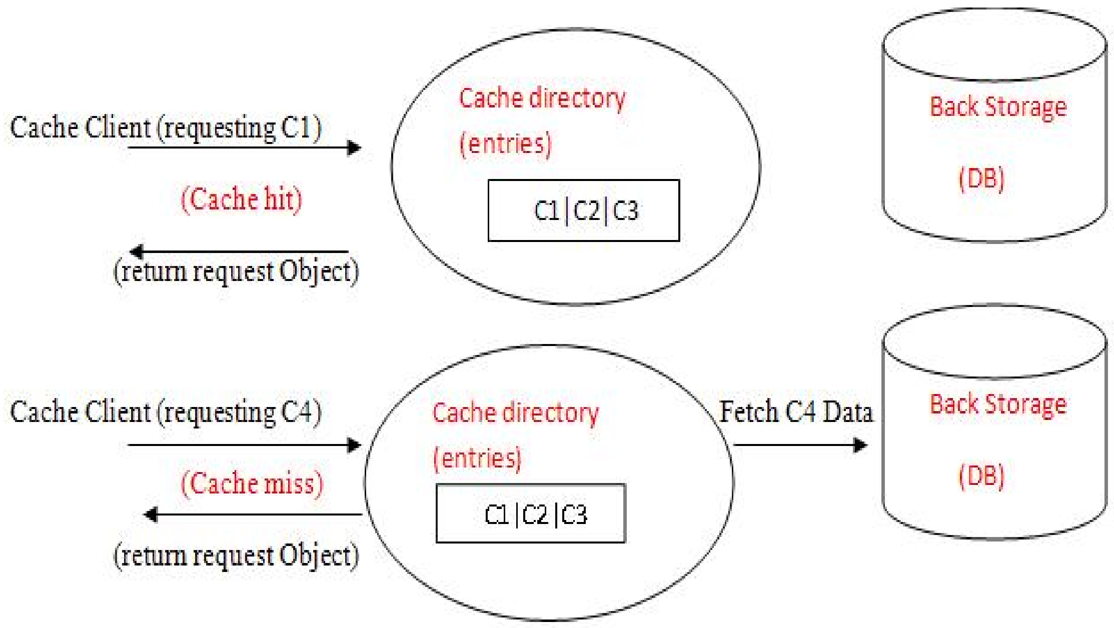


Figure 22 - Basic Caching Structure (from [8])

Ehcache – Ehcache is an open-source Java distributed cache for familiar purpose caching, J2EE and light-weight containers tuned for large size cache objects. It features memory, off heap store and disk store. Ehcache also acts as a pluggable cache for Hibernate, Spring and JPOX etc. The cache behaves as a local copy of data retrieved from or stored to the system-of-record (SOR). SOR is assumed to be a database in Ehcache [9].

Data stores which are promoted by Ehcache include:

- On – Heap Store – Utilizes Java's on-heap RAM memory to store the cache entries
- Off – Heap Store – It is limited in size only by the available RAM
- Disk Store – Utilizes a disk (filesystem) to store the cache entries [32]

The Figure 23 displays the storage tiers of Ehcache. Example when a cache is configured to use multiple data store, they are referred as tiers [32]. The functionalities of the storage tiers of Ehcache is explained below:

- Applications may have one or more Cache Managers
- A cache manager can manage multiple Caches
- Caches are configured to utilize one or more tiers for storing cache entries
- Ehcache stores the most recently accessed data in the typically less abundant tiers and the less recently accessed data in the more abundant tiers

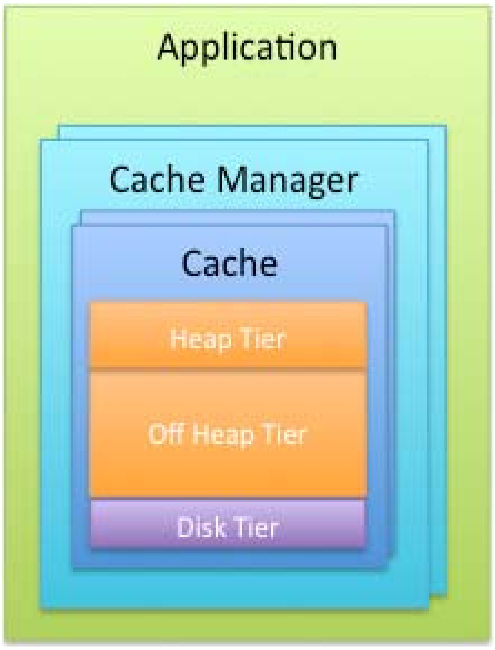


Figure 23 - Storage Tiers of Ehcache (from [32])

After the clear analysis and the sample code implementation of the Ehcache, the performance inefficiency is low in terms of speed, i.e. very slow when compared to JCS in putting and getting data from the disk and also the memory consumption is high. According to the reference [10] [11] the practical results of time consumed to get and put data on the disk with comparison to JCS and MapDB are proven.

Java Caching System (JCS) – JCS is also a distributed caching system written in Java for server-side Java applications where it is managed by Apache Software Foundation. It is proposed to accelerate the speed of applications by contributing a means to manage cached data of various dynamic natures. The basis of JCS is the Composite Cache, which is the pluggable controller for a cache area. Four different category of caches that can be plugged into the Composite Cache for any given region are [33]:

- LRU Memory Cache

- Indexed Disk Cache and JDBC Disk Cache
- TCP Lateral Cache
- RMI Remote Cache [33].

JCS uses a concept of regions that can be seen as cache instances. Each region can be deployed with different combinations of plugins adding different levels with specific behaviors to the instance [24].

According to the reference [28] the comparison between the JCS and LinkedHashMap is executed. The test results to of JCS vs LinkedHashMap states that JCS is too slow in both putting and getting the data from the cache. The test results depict that JCS is too slow when compared to the speed of HashMap which led to the selection of different approach to satisfy the thesis goals.

MapDB – MapDB is an embedded database engine for Java. Better performance is the result of compromises between consistency, speed and durability. MapDB provide distinct options to make various compromises. There are several storage implementations, commit and disk sync strategies. It provides collections backed by on-disk or in-memory storage. Data is stored in MapDB as a key value pair. By default, MapDB uses generic serialization which can serialize indiscriminate cast of data. It is agile and higher in memory efficiency to use specialized serializers. MapDB is adaptable and can be used in mutiple roles such as follows [13]:

- Drop-in replacement for Maps, Lists, Queues
- Off-heap collections not affected by the Garbage Collector
- Multilevel cache with an expiration and disk overflow
- RDBMs replacement with transactions, incremental backups
- Local data processing and filtering. MapDB can process the huge qualities of data in acceptable duration of time

There are two classes that act like the inseparable between the different pieces, namely the DBMaker and the DB classes [19].

The DBMaker class handles various functions like database configuration, creation and opening. MapDB has several modes and configuration options. Most of these can be set using this DBMaker class [19].

A DB instance illustrate an unclosed database (or a single transaction session). These DB instance can be utilized to create and open collection storages. These instance can also manage the database’s process with methods such as commit(), rollback() and close(). Where commit() function makes the changes made to the DB permanent, rollback() discards all the changes made within current transaction and close() just closes the opened DB and it is very necessary to call close() function to protect files from data corruption. These MapDB file storage can only be opened by a single user at one perticular time. File lock will prevent files being used multiple times [19].

To open (or create) a store, must use one of the DBMaker.xxxDB() static methods. MapDB has much higher count of formats and modes, whereby each xxxDB() uses distinct modes: memoryDB() which initiate an in-memory database backed by a byte[] array, appendFIleDB() opens a database which utilize append-only log files and so on [13].

An xxxDB() method is followed by one or more configuration options and finally a make() method which applies all options, open the selected storage and returns a DB object [13].

MapDB extracts HashMap and HashSet collections from HTreeMap. HTreeMap is endorsed for managing huge number of key/values. Java HashMap is a data structure, based on hashing which allows storing an object as a key - value pair and the HashMap class implements the interface Map<K, V>. The main methods of this interface are: [15]

- V put(K key, V value)
- V get(Object key)
- V remove(Object key)
- Boolean containskey(Object key)

And HashSet class implements the Set interface, backed by a hash table (actually a HashMap instance) [16] and it stores objects (elements or values). It has a great performance with large keys [23].

HTreeMap is a thread-safe and scales under parallel update [20].

- HTreeMap allow parallel writes by utilizing multiple segments, each with distinct ReadWriteLock. HTreeMap is a segmented Hash Tree
- HTreeMap will not utilize any firm size Hash Table which are used by other HashMaps, and does not rehash the whole data while Hash Table grows
- HTreeMap doesn’t need resizing of Index Tree as it uses auto-expantion of Index Tree
- HTreeMap also involve very definite amount of storage place, hence the empty hash slots do not utilize any space

MapDB can be utilized for several common use cases and obstacle. MapDB offers a very natural way to access the huge amount of data stored in a very agile paradigm, with a schema that precisely matches application criteria. MapDB also answer the issues of applications which suffer in running out of Java heap memory, or enormous Garbage Collection from pursuing to overflow with too large number of objects into the application runtime. [14]

3.2 GEMMA Architecture Analysis

The GEMMA architecture explained in the second chapter was analyzed to find the limitations in it as it was imposing problems for serialization. This architecture analysis gave rise to a new GEMMA architecture with new concepts that had to be implemented which is explained in the coming chapter. The main reason for serialization problems was that GEMMA only provided an interface IMappable that was implemented by each parser individually. It represents that there was no way to predict how the internal structure of these mappables looked like. These interfaces gave rise to objects of different kinds which increased complication for the serialization of data. As it’s analyzed that the data in GEMMA old architecture gets stored in distinct components, so the exporter needs to fetch the data from multiple locations. This data storage in multiple location complication needs the structuring of data storage necessarily so that the exporter obtains all the data from the single component.

So in the new architecture, the main aim is to get rid of this problematic IMappable interface by replacing it with a common Class Mappable that has to be used by all parsers and additionally a new component DataManager that stores the GEMMA data centrally and provides it to other components.

3.3 Selection of Desired Approach

3.3.1 Selection of Kryo for Session Data Storage

The research was first made on the choice of the Kryo serialization. Proper analysis and comparing results of other serialization gave way to choose Kryo. The main aim of the thesis was to store and restore the session data to avoid re-computing of huge mapped data each and every time the GEMMA tool is started. To solve this storing and restoring session data problem, kryo serialization is chosen. Where the serialization technique involves serialize session data to store the data on the disk and again deserialize this session data to retrieve back the stored data whenever needed.

3.3.2 Selection of MapDB for Reducing Memory Footprints

The second main goal of the thesis was to reduce the memory footprints during the runtime, to solve this problem MapDB database engine has been chosen. According to the research made and also the comparison between few other approach, MapDB solution is chosen for implementation in GEMMA project. The main technique involved in MapDB is storing the huge data on the disk instead of memory to reduce memory footprints. The MapDB database engine was selected to store the data which will be in terms of database (DB) files on the disk.

Chapter 4

4 Concept Introduction of GEMMA

The previous chapter explained the analyzed results of the old GEMMA architecture and the identified solution against the established criteria which seemed to be more beneficial. This chapter has two sections which will explain the concept of the new GEMMA architecture and the implementation of the chosen approach like Kryo and MapDB. The subchapters give the clear illustration about the new architecture and its benefit towards the desired goal to be achieved.

4.1 Change of GEMMA Architecture

The provident analysis of the GEMMA old architecture persuades the changes to progress in the new GEMMA architecture to reach the expected goals of the thesis. The disadvantage of the old GEMMA architecture made way for the new GEMMA architecture to get structured in an efficient manner to achieve the desired goals of the thesis.

The Main Concept was to get rid of Mappable interface in favor of Mappable class that is used by all parsers by including a new component called Data Manager in the existing GEMMA architecture. This Data Manager would be the centralized data management for the whole GEMMA project where it stores all the mapping data which is implemented by MapDB database engine which was chosen as a solution to reduce memory footprints during runtime. Data manager as a new component as an inclusion to the new GEMMA architecture which store the complete mapping data which was stored in multiple components in old GEMMA architecture. So, the three main concepts derived from all the analysis made here:

- To replace the IMappable interface causing serialization problems by the common class Mappable
- Implement data persistence for the overall project state through serialization/deserialization by Kryo
- To reduce the memory footprint during runtime by storing the data on the disk using MapDB

The above three concepts, have to be implemented to reach the goals set for the improvement in the GEMMA framework. The centralized Data Manager plays a greater role in managing the data and data storage in a precise manner. The figure 24 portrays the new GEMMA architecture with the Data Manager as a new centralized component.

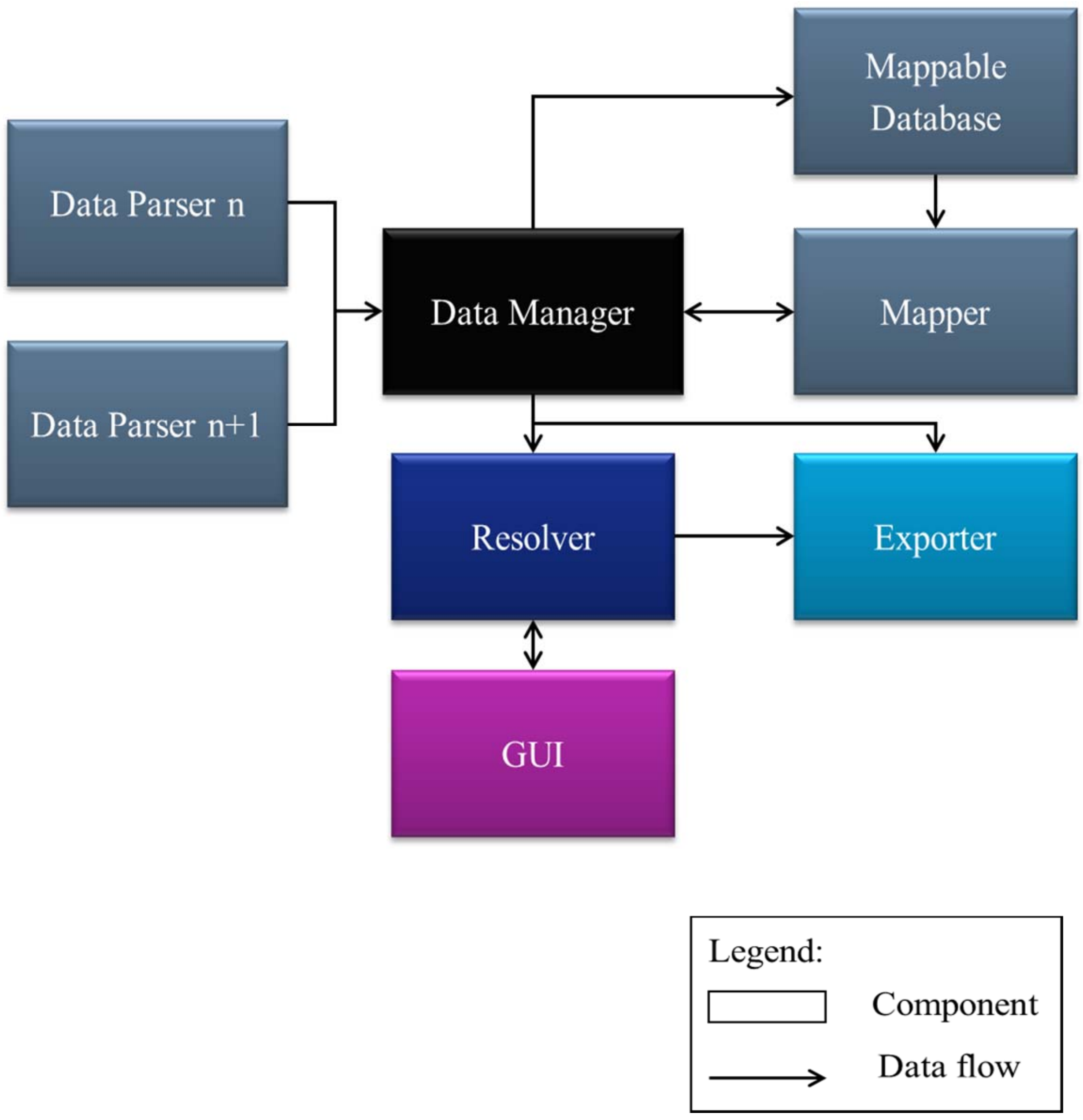


Figure 24 - New GEMMA Architecture

The new GEMMA architecture provides a few transformations in the data flow with the new centralized Data Manager and their categorization is interpreted as follows:

- One or more application-specific Data Parser reads data, convert it into the form of nodes and/or mappables and provide it to define a new data parser. All available parsers in GEEMA are registered in an internal parser registry where the Run Configuration can instantiate, configure and run those parsers, those are required by the configuration file. The data are then transfered to the Data Manager

- The Data manager will then transmit the mappable data to the mappable database i.e. the Lucene Database

- To be stored in the Lucene Database, all relevant information from a mappable must be converted into strings. Each mappable is assigned a unique ID from its parser and other required information is stored as a detail - value pair in so called details as shown by Figure 25

- Now Run Configuration runs in the background to just hold the configurations that define which parsers, exporters and rules are in current project instead of providing exporter with mappables, nodes and mapping data from data parser and mapper

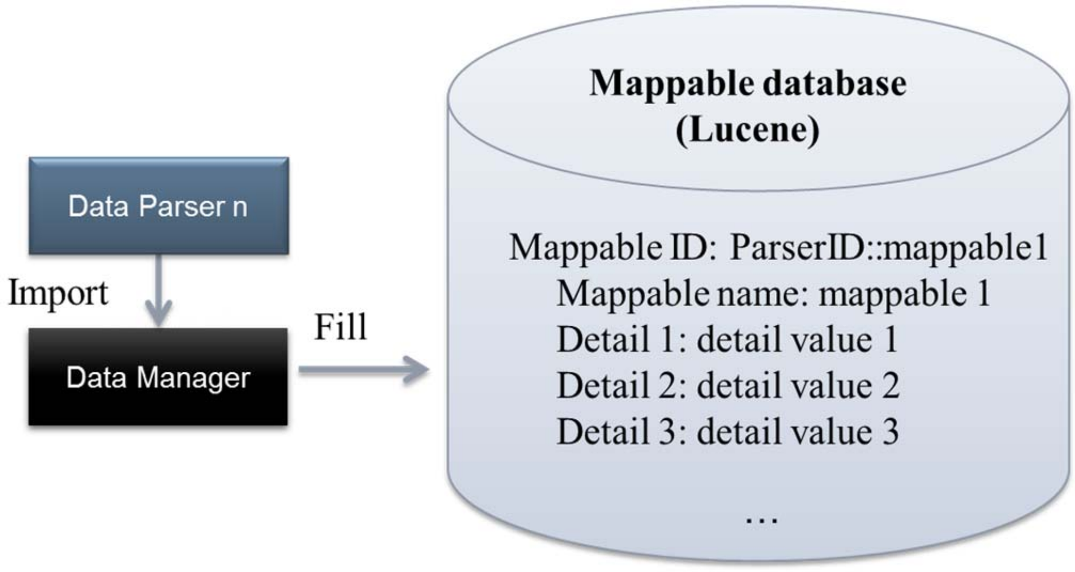


Figure 25 - Import to Lucene Database

- Mappables in the Mappable database are selected based on mappable details that will require matches. The required matches for each mappable that is selected for mapping are obtained by running queries on the mappable database. The queries are derived from the mapping rules which are user-defined. The set of rules which should be applied in one mapping project will be defined by the configuration

- As illustrated by Figure 26, during the matching process, the generic Mapper requests the Data Manager for existing mappables and receives it. For each and every mappable the mapper requests the applicable user rules from the rule manager and utilizes the content obtained in these applicable rules to generate a query that can be used to query the mappable database

- The results from the query are then utilized to generate a mapping from the original mappable to the query results. These mappings are stored in the Data Manager. A mapping is a one to many relations between one mappable and all the various matches that were found

- Resolving the matches is an alternative step that is highly driven by the specific application. It potentially requires the interaction with the user to make a specific selection, e.g., a mapping rule might say that for a mappable only one-to-one mapping is acceptable, but if more than one match was found then the user can decide which of the very nearest matches should be selected or not

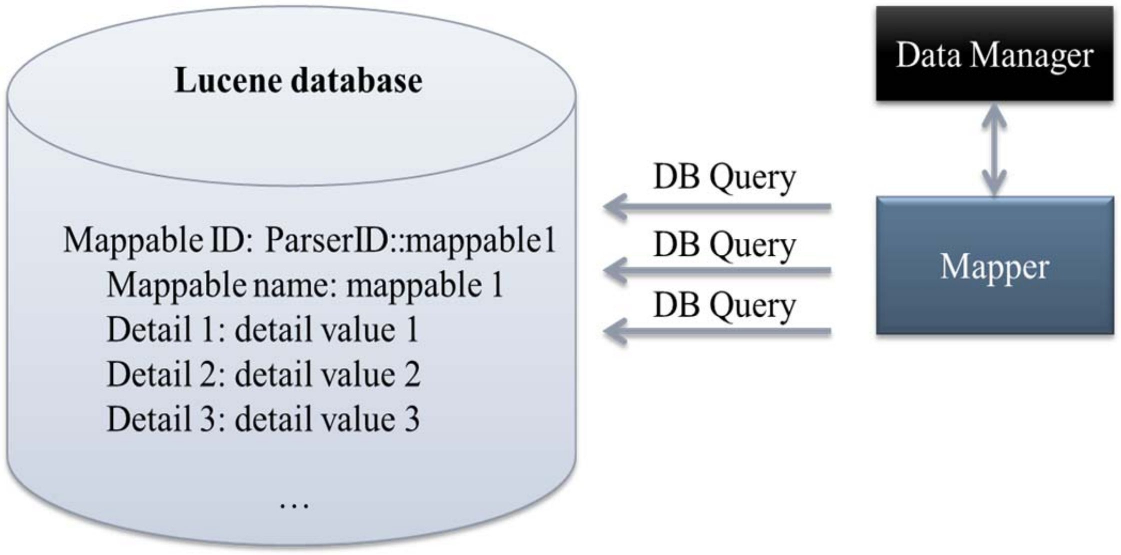


Figure 26 - Matching Process

- The Resolver also permits the user to implement the graphical user interface to review and validate the generated mapping results, and thereby to analyze the completeness and correctness of the defined user rules and to manipulate mappings manually, e.g., to eliminate a mappable from a mapping if the match was incorrect

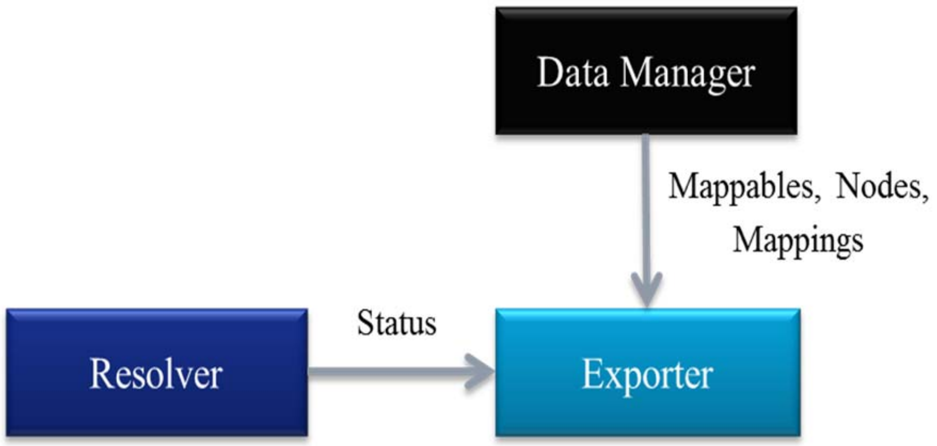


Figure 27 - Export Process

- At last is the Exporter which is a highly application specific. Where exporter task involves the transformation of the internal data model into an application-specific output file. Similar to the DataParser interface, a generic MappingExporter interface allows the definition of the custom exporters that are registered in an exporter registry where they can be accessed by the run configuration as dictated by the configuration file

- Each exporter can obtain the available mappables, nodes and mappings from the Data Manager as the new GEMMA architecture establishes the change in data storage with the centralized data manager. And the resolver as usual provides an exporter with the status of the data elements as depicted by Figure 27

- Using all the information provided by the Data Manager and Resolver the exporter creates a mapping export. The mapping export can take many forms, e.g., it can be just an XML file as the standard exporter produces, but it can also be an export directly into an application using the application's API. How the data is exported is completely encapsulated in the exporter

Module	Description	Core	Specific
Data Parser	Reads data (nodes and/or mappables) into the internal data model and feeds the mappable database		x
Mapper	Generates mappings between mappables based on rules	x	x
GUI	Interface for loading configuration, displaying mappings as well as allowing user-decisions and displaying of data based on resolver as shown in Figure 2.9	x	
Mappable Database	Stores mappable information and allows searches	x	
Data manager	Stores mappables, nodes and mappings	x	
Resolver	Resolves mappings based on application specific semantics		x
Run Configuration	Holds the configuration that defines which parsers, exporters, mapper and rules are used in the current mapping project	x	
Exporter	Exports the internal data model into a specific file format		x

Table 2 - Generic Mapping Framework Modules (from [1])

By the detailed description of the new GEMMA architecture which shows how the new Data Manager component brings an advancement when compared to old GEMMA architecture. The new architecture is the baseline for the implementation of the data manager in the GEMMA project as a Java code as the complete implementation of GEMMA tool was done in Java on top of Eclipse Rich Client Platform (RCP). This new GEMMA architecture makes the implementation of the most beneficial solution chosen for the established criteria evident. In further there would be a detailed description of the new approach implementation.

4.2 Implementation of the New Approach

The technical design and implementation was the most crucial part of the thesis work. In this section, the implementation of the code generated in MapDB for the data storing and restoring on-disk and Kryo serialization for session data storage is discussed in detail.

MapDB Code Implementation

MapDB is an open-source, embedded Java database engine and collection framework [19]. MapDB is flexible with many configuration options. But in most cases, it is configured with just few lines of code. MapDB was chosen to be the solution to store complete data on disk instead of ram for reducing memory footprints during runtime. As explained in section 3.1 MapDB seemed to exhibit consistently efficient performance compared to other selected approaches examined.

Installation

The installation of MapDB to the existing GEMMA project is by downloading a single MapDB jar file directly from the Maven Central repository and add the downloaded mapdb-3.0.0-M6.jar file to the GEMMA project classpath. The MapDB 3.0 version is used in this current thesis work. The new DataManager class package was plugged into the existing GEMMA Java project to build a new class in it for the Data Manager. There are few terminologies and unique functions in MapDB to be familiarized, so the following detailed description.

DB and DBMaker

MapDB is a set of coupled components. To hold things together, there are two main classes, namely DBMaker and DB as explained in section 3.1. Where

- DBMaker handles database configuration, creation and opening
- DB represents opened database (or single transaction session)

So to start coding in MapDB the first step is to create a store, once it's created all the upcoming processes can just close or clear the store and use as per required. fileDB() method is followed by few configuration options and make() method which implies all options, open storage and return DB object. As the chosen solution is to store data on the disk, based the

MapDB program code in appendix A the db file named mappable.db, node.db and mapping.db is created to stored respective data on the disk.

The configurations used in appendix A in the fileDB() method are:

- **fileMmapEnable()** – Memory mapped files are activated with this setting. Memory mapped files (mmap) is a fastest storage option for disk storage in MapDB when compared to Random Access File (RAF) and File Channel. The exception {@code java.lang.OutOfMemoryError: Map failed} is only on 32bit JVM, if this mode is enabled. Memory mapped files are highly dependent on the operating system [21]

- **fileMmapEnableIfSupported() –** Enable Memory Mapped Files only if the current JVM supports it (is 64 bit)

- **fileMmapPreclearDisable()** – To make the memory mapped files faster this mode is called

- **cleanerHackEnable() –** Closes the file on db.close(). This mode is called because of a bug encountered in a JVM on Windows. The memory mapped files were remained open even after db.close() was called. This mode now prevents file to be reopened or deleted on Windows

- **closeOnJvmShutdown() –** Adds JVM shutdown hook and closes the DB just before the JVM [22] just to protect data if the JVM crashes or is killed [21]

Hence the dbMappableDisk, dbNodeDisk and dbMappingDisk are created similarly with the above set of configurations.

Open and Create Collection

Once the DB is created it can open a collection or other records. DB uses builder style configuration. It starts with the type of collection (e.g., HashMap, treeSet…) where HashMap is the type used in this thesis, and name followed by the configuration is applied and finally by operation indicator [19].

According to appendix A, the following code snippet opens (or creates new) 'onMappableDisk' named record

```
onMappableDisk = (HTreeMap<String,Object>)dbMappableDisk.hashMap("onMappableDisk")
                 .keySerializer(Serializer.STRING)
                 .hashSeed(111)
                 .createOrOpen();
```

HTreeMap has a number of parameters. The most prominent parameter is **name**, which determines a Map within a DB object and **serializers** which manage data inside the map. MapDB utilizes Key Serializer to create Hash Code and to compare the keys [20].

- Serializer.STRING uses stronger XXHash which generates less collisions [20]

- HTreeMap adds hashSeed(111) for protection against Hash Collision Attack. These Hash Collision attacks are randomly generated while the collection is produced and persisted together with its description. User can also supply own unique Hash Seed

The similar configurations of onMappableDisk are used to create or open the collection in MapDB code for onNodeDisk and onMappingDisk as in appendix A. The builder can end with three different methods such as:

- **create()** – It will create a new collection, and throws an exception if the collection already exists

- **open()** – It opens an existing collection, and throws an exception if it does not exist

- **createOrOpen()** – It opens an existing collection if it already exists, or else creates the new one. This is the most adaptable option to be used in the MapDB as displayed in appendix A

Transactions

DB has few methods to handle a transaction lifecycle in particular manner: commit(), rollback() and close(). One DB object represents single transaction.

- **commit() –** It persists all the changes made by the user to the disk

- **rollback() –** It discards all the changes previously made by the user within the current transaction

- **close() –** It closes the store completely

To protect the file from corruption, MapDB offered Write Ahead Log (WAL) to make file changes atomic and durable. But WAL is slower as data has to be copied and synced multiple times between files, so WAL is disabled by default [21]. With this WAL disabling there would be no protection against a store or JVM crash. In this case the store must be closed correctly or else there would be data loss. If MapDB detects an unclean shutdown, it refuses to open such corrupted storage [21]. During this WAL disabling, the rollback function throws an exception, so commit function attempts to flush all the write caches and synchronizes the storage files [21]. So once the commit function is called even when there are no writes, the store would be secure (no data loss) in case of JVM crash. The example below is a code snippet to explain the above methods which uses a single transaction per store.

```
ConcurrentNavigableMap<Integer,String> map =
db.getTreeMap("collectionName");
map.put(6,"six");
map.put(7,"seven");
//map.keySet() is now [6,7] even before commit
db.commit();  //persist changes into the disk
map.put(3,"three");
//map.keySet() is now [6,7,3]
```

```
db.rollback(); //reverts recent changes
//map.keySet() is now [6,7]
db.close();
```

According to the MapDB code in appendix A, the commit() is called to persist all the changes made to mappables, nodes and mappings to the disk before closing the stores respectively in shutdown() method. The MapDB code in Appendix A consists of two similar kinds of methods called shutdown() and close1(). Where shutdown() method is called in RunConfiguration class of the GEMMA project to persist the configured data and close the store before reopening of the restored data from the disk. And close1() method is called in the GEMMAMainPart class to just close the mappable, node mapping stores before starting the new GEMMA application which could lead to store corruption and JVM process crashes if the stores are not closed properly. And onMappingDisk.clear is called to delete all the mapping data on the disk.

As already described MapDB uses generic built-in serialization to serialize whole data and store it on the disk, this thesis work also implements MapDB using kryo serialization. The complete examined results of both MapDB using built-in serialization and MapDB using kryo serialization against old GEMMA tool efficiency are explained in chapter 5.

Kryo Serialization

Kryo serialization is a fast and efficient object graph serialization framework for Java. Kryo is used to persist data objects to a file, database, disk or even over the network. Kryo does not enforce a schema or concern about what data is written or read. This is left to the serializers. Appendix B is the Kryo serialization code which is implemented to store the session data as a .gma file format on the disk during runtime.

Installation

Kryo is used without maven by adding the jar files to the GEMMA classpath. As kryo jar has a some external dependencies, they are:

- **MinLog logging library** – It is a tiny logging library which has a few features like zero overhead, extremely lightweight, simple and efficient at runtime [25]
- **Objenesis library –** It is a library dedicated to bypass the constructor when creating an object [26]
- **ReflectASM library –** It is a very small Java library that provides reflection by using code generation [27]

All these three jar files along with kryo-3.0.3. jar file is added to the GEMMA classpath before kryo code implementation.

IO

The Input class of a kryo serializer is an InputStream that reads the whole data from a byte array buffer. This byte array buffer can be set directly, if the reading from a byte array is desired. If the Input is provided to an InputStream, it will fill up the buffer from the stream when the buffer is disabled. Input has numerous methods to efficiently read the primitives and strings from bytes. It produces functionality which is similar to DataInputStream, BufferedInputStream, FilterInputStream and ByteArrayInputStream [4].

According to Appendix B, which depicts that the session data is read from the disk, i.e. to deserialize or restore the data stored in the form of a file on the disk, FileInputStream is used.

The OutputStream is an Output class that writes the whole data to a byte array buffer. This buffer can be obtained and utilized directly, if a byte array is desired. While the Output is given an OutputStream, it will expel the bytes to the stream when the buffer is completely filled. Output has multiple methods for efficiently writing primitives and strings of bytes. It provides functionality similar to a DataOutputStream, BufferedOutputStream, FilterOutputStream and ByteArrayOutputStream [4].

The Output buffers when writing to an OutputStream, flush() or close() function must be called after writing is complete so that the buffered bytes are written to the underlying stream.

According to Appendix B, to write the data on the disk, i.e. to serialize the session data and store it on the disk in the form of data objects, FileOutputStream is used.

Finally the data is serialized compressed into a zip file as .gma format and stored on the disk and the deserialization involves unzipping the file later deserializing and restoring back to GEMMA framework when called by the user.

By this the implementation processes of MapDB and Kryo for the desired goal of the thesis are completed and the following chapter discusses in detail about the careful examined results.

Chapter 5

5 Results, Discussion and Future Work

In this chapter, an overview of the GEMMA tool with the MapDB implementation results are discussed. This chapter also presents the performance trials conducted with multiple tests in terms of memory consumption and the time taken by the GEMMA tool to complete the mapping task are generated. It aims to display the effectiveness of the new approach MapDB and Kryo serialization by comparing different approaches to achieve the same goal. The areas of application of the code generated and the future extension of this thesis of the new approach is briefly explained.

5.1 Java VisualVM Testing Tool

The testing tool used to calculate the memory consumption of each GEMMA mapping process was VisualVM which is by default wrapped in the JDK. VisualVM is really simple and yet powerful to find the memory usage of the Java application during runtime. The VisualVM application monitors and troubleshoots applications running on Java using various technologies including jvmstat, Serviceablility Agent (SA) and Attach API [29].

Display local and remote Java process

VisualVM automatically detects and lists locally and remotely running Java applications and applications can also be manually defined by using JMX [2.9].

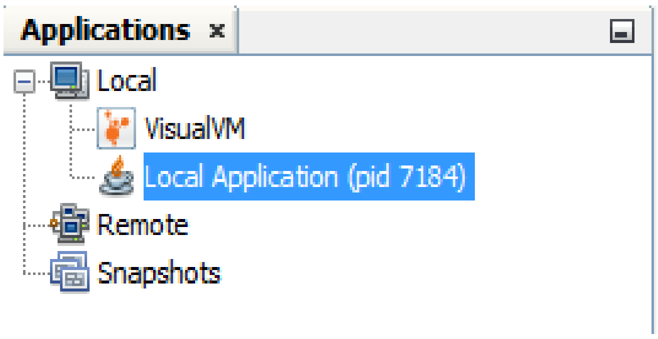


Figure 28 - VisualVM Application GUI

There are five different tabs in VisualVM like overview, monitor, threads, samplers and profiler. The **overview tab** displays the information about the launched Java application.

The next tab is the **monitor tab** in Figure 31 which illustrate the CPU and memory usage of the application. Monitor tab contains four different graphs. The first graph monitors application CPU usage and garbage collector activity, second graph monitors heap and metaspace / permanent generation memory, third graph monitor the number of loaded classes and fourth graph monitors the running threads [29].

Monitor process performance and memory consumption

The Figure 29 displays the heap graph which is one of the graph in monitor tab. This heap graph was mainly used to record each GEMMA application memory consumption in both old GEMMA and MapDB implemented GEMMA application. This graph was the main focus in the whole VisualVM application of the thesis.

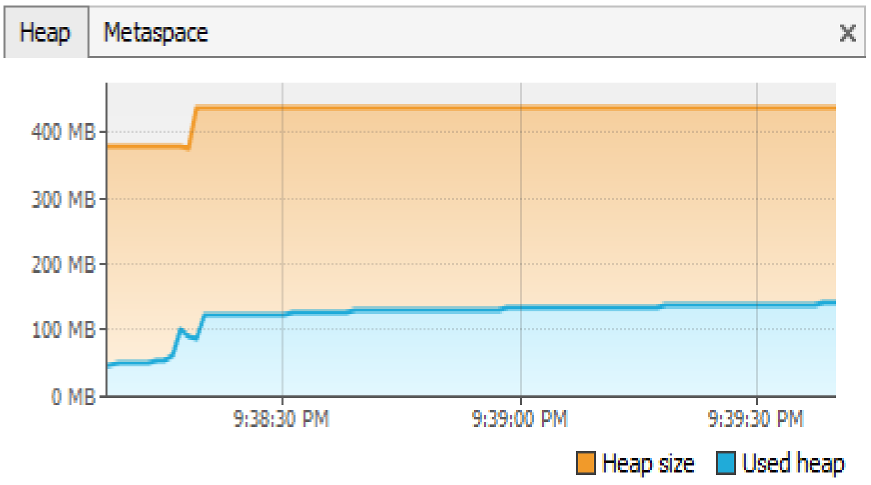


Figure 29 - Heap Graph in VisualVM

Visualize process threads in threads tab

The third tab in the VisualVM is the **threads tab**. All the multiple threads running in a Java process are presented in a timeline together with aggregated Running, Sleeping, Wait, Park, and Monitor times [29].

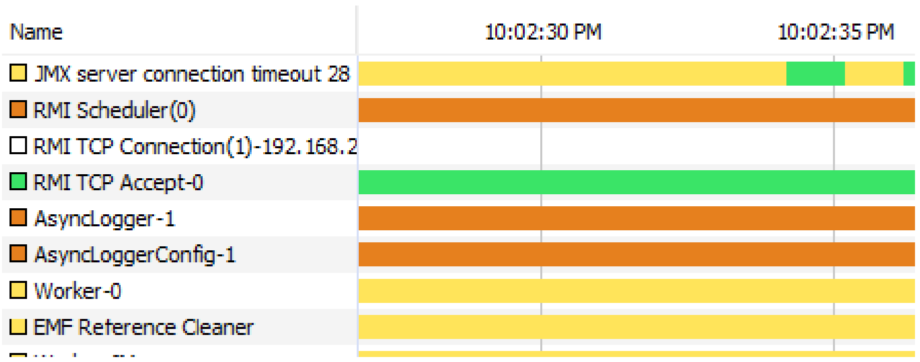
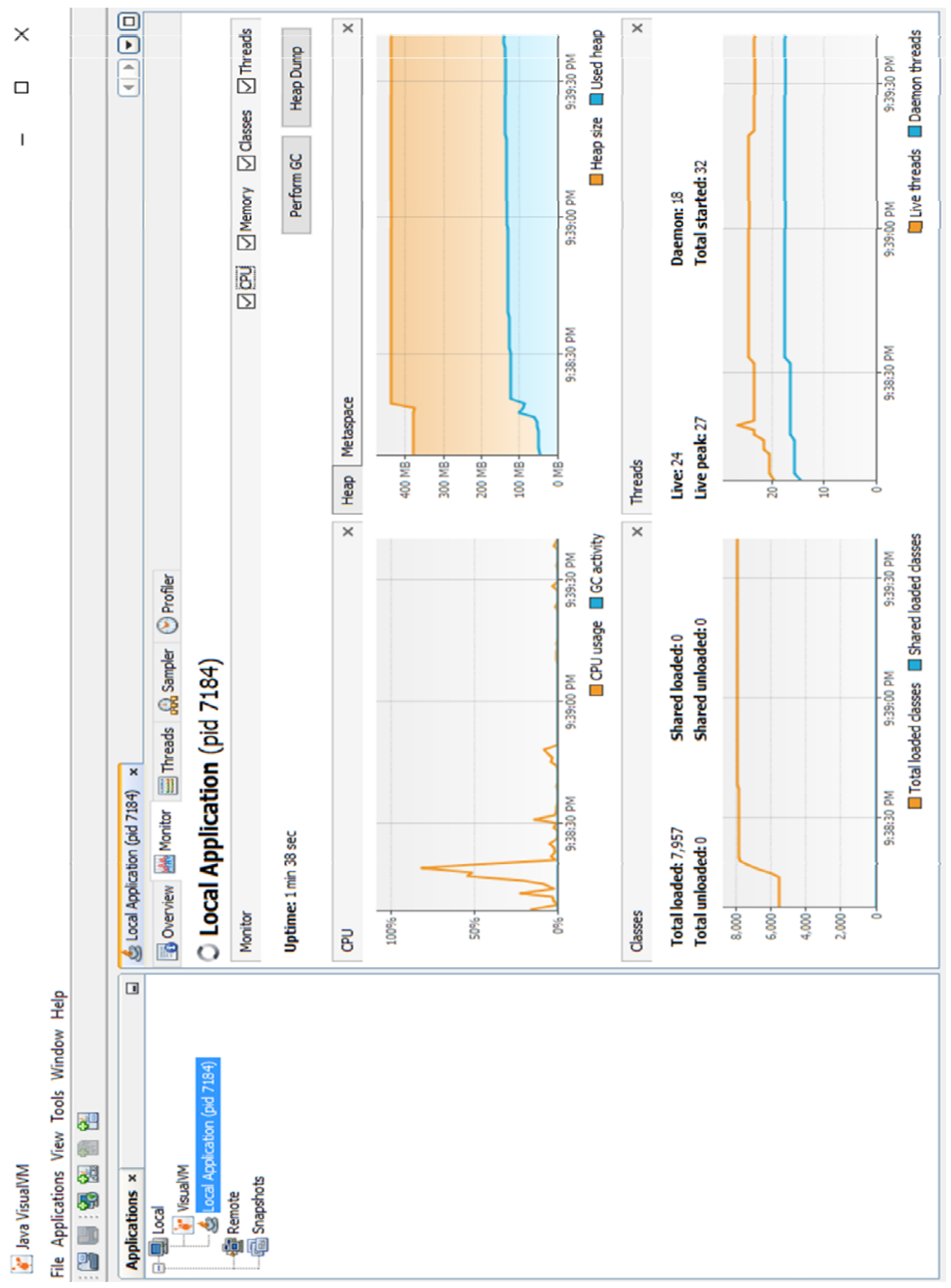


Figure 30 - Threads Tab in VisualVM

Profile performance and memory usage

VisualVM provides basic profiling capabilities for analyzing application performance and memory management [29]. This profiling capability is obtained by the fourth tab of VisualVM which is the sampler tab, it regularly samples the Java application in terms of CPU sampling to collect performance data and memory sampling to collect memory data and by fifth tab of VisualVM which is profiler tab to instrumenting the classes and methods of Java application.


Local Application (pid 7184)

Uptime: 1 min 38 sec

Total loaded: 7,957
Total unloaded: 0
Shared loaded: 0
Shared unloaded: 0

Live: 24
Live peak: 27
Daemon: 18
Total started: 32

Figure 31 - VisualVM GUI of Monitor Tab

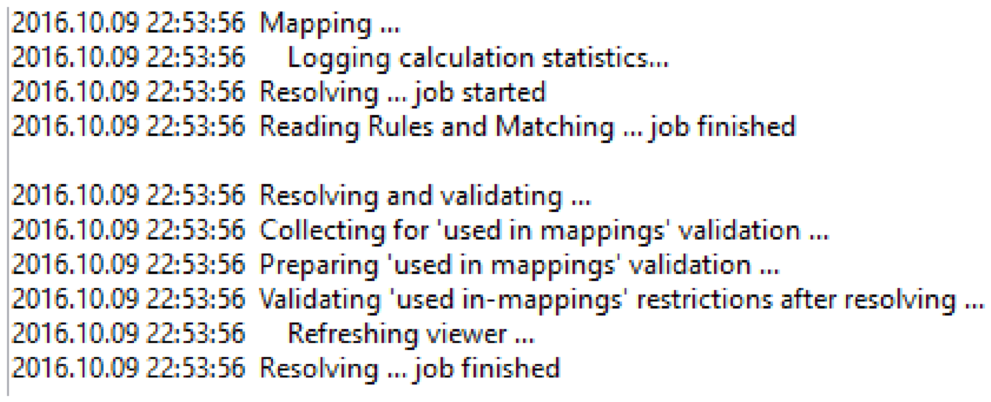
5.2 Testing and Validation

The result of new concept implementation of the existing GEMMA tool is tested and recorded. The kryo implementation of the GEMMA session storage is to avoid the recomputing of data every time when the GEMMA is newly launched. The implementation is now used to save the project, on the disk and also retrieve back. The serialized data is stored on the disk as a zip file to reduce the size of the file and this zipped file is unzipped and deserialized when retrieving the saved data, which is all implemented using kryo serialization.

MapDB data engine implementation in the GEMMA framework to offload data from the RAM to the disk storage is successfully implemented with all the interconnections to be made internally in the existing GEMMA framework. The testing of this new approach implementation involves multiple data strength mapping and trials based on the GEMMA particular memory consumption. The comparison between the new GEMMA framework with MapDB implementation and the old GEMMA framework is made on performance gain in terms of speed and memory consumption.

The main comparison was based on the speed of data computation in mapping in terms of time. The two main regions where GEMMA claims time to execute large data are during mapping and resolving the data. The amount of data depends on the combination of number of mappables and nodes in one particular mapping project. The time consumed for mapping involves establishing connections between the selected mappables based on user defined rules. And the time consumption of resolving process involves resolving and validating the mappings and also adding the validation status of mappables to index. This mapping time and the resolving time vary on the total sum of mappables and node data used in one particular mapping project.

To analyze the effectiveness of the new implementation of the GEMMA framework against the old GEMMA tool, numerous test cases have been tested and reviewed. These test cases are first tested for the MapDB custom serialization using kryo and the second set of testing is on MapDB built-in serialization. The two sets of serialization version in MapDB are tested on three different memory limits, by limiting the Java heap to 512 MB, 1024 MB and 2048 MB respectively. The memory consumption by each mapping project is calculated by using Java VisualVM.



```
2016.10.09 22:53:56  Mapping ...
2016.10.09 22:53:56     Logging calculation statistics...
2016.10.09 22:53:56  Resolving ... job started
2016.10.09 22:53:56  Reading Rules and Matching ... job finished

2016.10.09 22:53:56  Resolving and validating ...
2016.10.09 22:53:56  Collecting for 'used in mappings' validation ...
2016.10.09 22:53:56  Preparing 'used in mappings' validation ...
2016.10.09 22:53:56  Validating 'used in-mappings' restrictions after resolving ...
2016.10.09 22:53:56     Refreshing viewer ...
2016.10.09 22:53:56  Resolving ... job finished
```

Figure 32 - GEMMA Console Tab

The testing is launched by illustrating the difference between Old GEMMA Mapping Time (OGMT) and New GEMMA Mapping Time (NGMT), Old GEMMA Resolving Time (OGRT) and New GEMMA Resolving Time (NGRT) measured in seconds and finally Old GEMMA Memory Consumption (OGMC) and New GEMMA Memory Consumption Time (NGMC) measured in megabytes (MB) for five selected set of iterations based on Number of Mappables and Nodes (NMN) i.e., 1000, 10000, 25000, 50000, 100000 to be precise in the variation. The testing was conducted with five trials of each number of mappables and nodes respective of the memory limits selected and the average of each trials are mentioned in the following tables. The mapping time and resolving time of GEMMA process was recorded from the GEMMA console tab as shown in Figure 32 which reveals the present application details by its prescribed time consumption.

The two sets of comparison between the old and the new GEMMA framework commenced from the kryo serialization and then followed by the MapDB built-in serialization. The Table 3, 4, 5 depicts that the new MapDB implemented GEMMA framework readings for NGMT and NGRT are too high when compared to the old GEMMA framework OGMT and OGRT. The Table 3 for memory limit of 512 MB illustrate that for the huge NMN value the old GEMMA framework breaks down during the mapping process without completion of the task and the new GEMMA framework. But the new GEMMA framework executes the mapping process and stops by throwing ‘GC Overhead limit exceeded’ error during resolving particular mapping project. The ‘GC Overhead limit exceeded’ error indicates that the garbage collector runs all the time and the GEMMA tool is making very slow progress. So in 512 MB memory usage the GEMMA tool shuts down without any good performance status even when MapDB is implemented.

The further investigation of the GEMMA MapDB performance results were not satisfactory with any particular performance in terms of speed or memory consumption. The NGMT seems to be acceptable as there is an increase of 20-30% of mapping time, but the NGRT is largely high in terms of hours for the 50000 – 100000 mappables and nodes. The same unsatisfactory results of MapDB using kryo serialization continue with the MapDB using built-in serialization as shown in Table 6, 7, 8. The main issue of MapDB kryo serialization was the NGRT error in 512GB memory usage and high time consumption during NGRT for 50000 – 100000 mappables and nodes in 512 GB, 1024 GB, 2048 GB which was still worse while using MapDB built-in serialization. Because when compared with Table 5 and 8 the NGRT and NGMC the built-in serialization is too high.

As the GEMMA MapDB time consumption during the resolving process of any mapping data was high when compared to old GEMMA resolving process, this lead to the further improvements in the MapDB code implementation to enhance the performance of MapDB during resolving data. The analysis leads to a few changes in the DataManager class and also in the GEMMA project for an effective result to achieve the desired goals. The following section 5.1.1 explains the analysis of MapDB code and the changes made.

Comparison of GEMMA MapDB Kryo Serialization versus Old GEMMA

Memory Limit – 512 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	4	1	6	233	158
10000	6	15	4	208	298	319
25000	15	30	11	1260	350	460
50000	38	54	16	3 Hrs	450	400
100000	Stops	118	Stops	Error	Stops	Error

Table 3 - GEMMA Module Trial Results

Memory Limit – 1024 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	3	1	5	260	288
10000	6	15	4	195	426	448
25000	13	27	11	500	525	635
50000	36	52	22	1 Hr	675	845
100000	82	112	50	10 Hrs	938	1100

Table 4 - GEMMA Module Trial Results

Memory Limit – 2048 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	3	1	6	233	432
10000	6	12	5	190	738	758
25000	14	29	11	1098	800	920
50000	37	53	18	1Hr	900	1100
100000	82	109	57	5Hrs	1260	1500

Table 5 - GEMMA Module Trial Results

Comparison of GEMMA MapDB Built-in Serialization versus Old GEMMA

Memory Limit: 512 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	4	233	170
10000	6	10	4	253	298	365
25000	15	16	11	1980	350	460
50000	38	49	16	3 Hrs	450	450
100000	Stops	116	Stops	Error	Stops	Error

Table 6 - GEMMA Module Trial Results

Memory Limit: 1024 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	4	260	223
10000	6	9	4	246	426	453
25000	13	18	11	1725	525	900
50000	36	49	22	2 Hrs	675	920
100000	82	112	50	12 Hrs	938	1000

Table 7 - GEMMA Module Trial Results

Memory Limit: 2048 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	4	233	585
10000	6	9	5	230	738	765
25000	14	15	11	1500	800	1000
50000	37	51	18	2 Hrs	900	1900
100000	82	115	57	9 Hrs	1260	2000

Table 8 - GEMMA Module Trial Results

5.2.1 Updated MapDB Code

The examination of the initial MapDB code for the improvements gave way for the identification of few corrections in whole GEMMA project. The main correction in MapDB code which is defined as a DataManager class in GEMMA Java project was in the methods getAllMappables() and getAllNodes(). These getAll() methods used to every-time read the whole GEMMA data every-time they were called. So when the huge data task is executed there would be large time consuming by the GEMMA tool to resolve the data.

The following code snippet below was the getAll() method used to get all the nodes in the DataManager class which used to read the whole node data every-time the GEMMA tool is executed.

```
public HashMap<String, Node> getAllNodes() {

        HashMap<String, Node> nodes = new HashMap<String,
Node>();
        for (Object key : onNodeDisk.keySet()) {

       nodes.put(((Node)onNodeDisk.get(key).getID(),
       (Node)onNodeDisk.get(key));

        }

        return nodes;

    }
```

The above code snippet was replaced by the code snippet below i.e., getAllNodes() method was replaced by getAllNodeKeys() method. The important modification in getAllNodeKeys() method was by only returning the set view of all the keys in the hashmap. This getAllNodeKeys() method reads only the updated data instead of reading the whole data every-time. The similar method was implemented to read the mappable and mapping data during the resolving process in GEMMA as shown in Appendix A.

```
public Set<String> getAllNodeKeys() {

        return onNodeDisk.keySet();

    }
```

The above new changes made in the MapDB code and also some other internal changes in the GEMMA whole project brought a drastic change in the time consumption. The complete updated MapDB using kryo serialization code used for testing is in Appendix C and MapDB using built-in serialization is in Appendix A.

Comparison of GEMMA Updated MapDB Kryo Serialization versus Old GEMMA

Memory Limit: 512 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	3	1	4	233	160
10000	6	15	4	23	298	289
25000	15	31	11	57	350	407
50000	38	58	16	115	450	482
100000	Stops	116	Stops	241	Stops	483

Table 9 - GEMMA Module Trial Results

Memory Limit: 1024 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	3	1	3	260	269
10000	6	15	4	22	426	430
25000	13	30	11	56	525	558
50000	36	56	22	114	675	701
100000	82	110	50	234	938	791

Table 10 - GEMMA Module Trial Results

Memory Limit: 2048 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	3	1	3	233	268
10000	6	14	5	22	738	681
25000	14	30	11	55	800	820
50000	37	54	18	114	900	978
100000	82	111	57	241	1260	1220

Table 11 - GEMMA Module Trial Results

Comparison of GEMMA Updated MapDB Built-in Serialization versus Old GEMMA

Memory Limit: 512 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	1	233	139
10000	6	8	4	7	300	257
25000	15	17	11	18	375	337
50000	36	47	22	39	490	454
100000	Stops	115	Stops	93	Stops	478

Table 12 - GEMMA Module Trial Results

Memory Limit: 1024 MB

NMN = No of Mappables and Nodes, **OGMT** = Old GEMMA Mapping Time, **NGMT** = New GEMMA Mapping Time, **OGRT** = Old GEMMA Resolving Time, **NGRT** = New GEMMA Resolving Time, **OGMC** = Old GEMMA Memory Consumption, **NGMC** = New GEMMA Memory Consumption

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	1	260	170
10000	6	8	4	7	426	382
25000	13	16	11	17	550	503
50000	38	51	22	39	700	641
100000	82	118	52	90	948	851

Table 13 - GEMMA Module Trial Results

Memory Limit: 2048 MB

NMN	OGMT (sec)	NGMT (sec)	OGRT (sec)	NGRT (sec)	OGMC (MB)	NGMC (MB)
1000	1	2	1	1	233	125
10000	6	8	5	7	738	650
25000	14	17	11	17	860	792
50000	37	50	23	39	960	900
100000	82	118	57	88	1270	1174

Table 14 - GEMMA Module Trial Results

5.3 Overview of GEMMA MapDB Solution

The results obtained from the updated code of DataManager class has been enhanced between the two sets of serialization testing when compared to the old GEMMA results. The main objective of the testing was to examine whether the chosen solution, MapDB is suitable to achieve the goals of the thesis. So the multiple test trials were executed to record the test results and ultimately aimed for the best results.

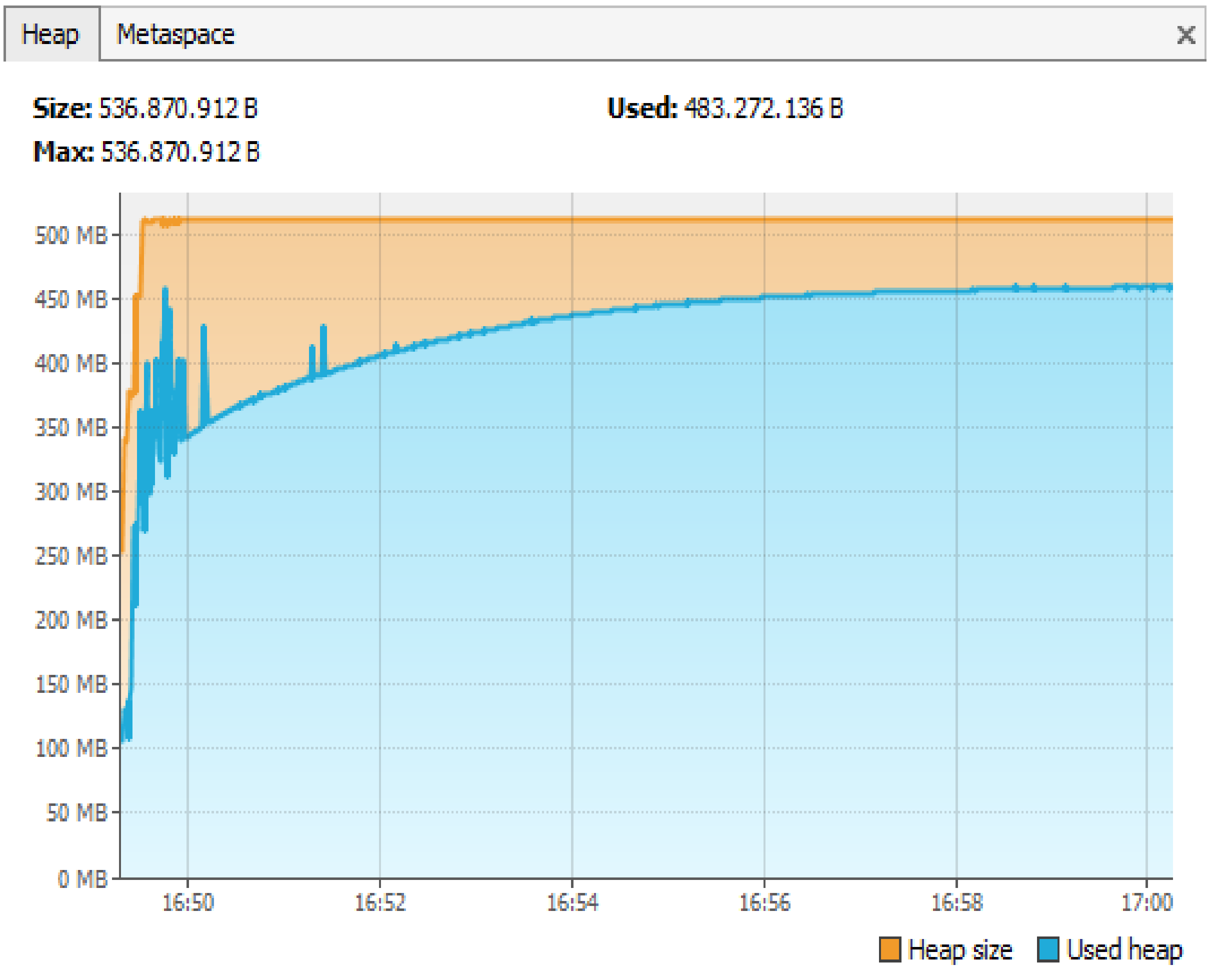
Again the same testing scenarios as in section 5.1 continued to examine the results brought by the changes made in the old code of DataManager class. The MapDB using kryo serialization results is noted down in the Table 9, 10, 11 and MapDB using built-in serialization results are listed in Table 12, 13, 14 for the clear observation. There was a drastic change in the results obtained by both the set of serialization patterns in MapDB when compared with the older version of the MapDB results.

This segment is the comparison of MapDB previous solution and the MapDB improved solution results. The NGRT of the MapDB kryo serialization with a 512 MB memory limit in Table 3 and Table 9 of same functionalities is considered as it had a massive drawback. The massive drawback was that the GEMMA process was incomplete because resolving the huge data of about 100000 numbers of mappables and nodes even after using MapDB as seen in Table 3 NGRT tend to be an error. The immense development with improved code is that the GEMMA function is completed even when the memory usage is limited to the least possible value. As seen in Table 9 the resolution of huge data of 100000 mappables and nodes is executed with the reasonable memory usage with 512 MB memory limit. Considering the Table 3 and 9 the NGMT and NGMC value remains almost similar but NGRT value of Table 9 is reduced by 70%-90% when compared to NGRT of Table 3 by solving the massive drawback in GEMMA.

Followed by the comparison of similar functionalities of MapDB kryo serialization of the previous and the present results with memory limit of 1024 MB and 2048 MB is considered in this particular division. Even with 1024 MB and 2048 MB results in the drastic reduction of 70% - 90% in the resolving time of the GEMMA process, i.e. NMRT of Table 10 and Table 11 when compared to NMRT of Table 4 and Table 5 like as in the 512 MB memory limit. Along with the resolving time the memory usage during the GEMMA process, i.e. NGMC of Table 10 and Table 11 has a slight reduction of 10% - 30% when compared to NGMC of Table 4 and 5.

Considering the comparison between the pervious results of MapDB built-in serialization in Table 6, 7, 8 and Table 12, 13, 14 which is the present results obtained with the improved code in MapDB built-in serialization. The comparison results imply that the best results are obtained with the improved code of MapDB built-in serialization with any memory usage limit. The improved MapDB built-in serialization results with a huge reduction of resolving time by 90% - 97% and also the memory consumption by 10% - 20% of each GEMMA process when compared to the previous MapDB built-in serialization.

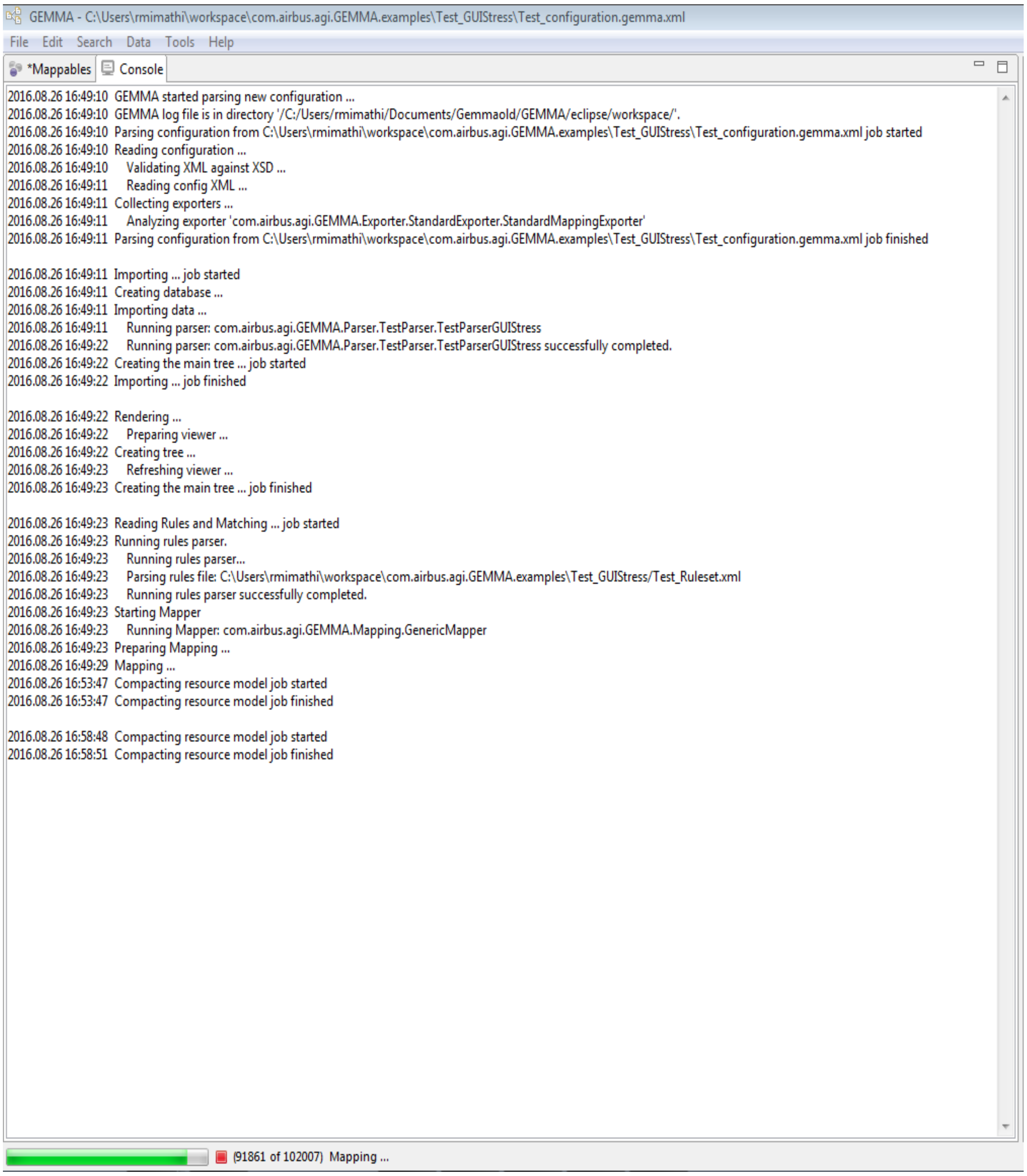
The Figure 33 depicts the visaulvm view of the old GEMMA application process breakdown while resolving 100000 mappables and nodes as a data. The memory limit is constrained to 512 MB the old GEMMA application halts without completion of mapping data and execution of the results.



Size: 536.870.912 B **Used:** 483.272.136 B
Max: 536.870.912 B

Figure 33 - Old GEMMA Breakdown Java VisualVM

The old GEMMA application breakdown when the huge data are used for mapping, this GEMMA breakdown in the initial stage of mapping is displayed in Figure 34 in the GEMMA console tab. Where the console tab depicts the complete information about the launched GEMMA mapping process. When the old GEMMA processed 100000 mappables and node data it was ceased without any further improvements as it was not efficient to compact the data in the given memory limit.


GEMMA - C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml
File Edit Search Data Tools Help
*Mappables
Console

```
2016.08.26 16:49:10  GEMMA started parsing new configuration ...
2016.08.26 16:49:10  GEMMA log file is in directory '/C:/Users/rmimathi/Documents/Gemmaold/GEMMA/eclipse/workspace/'.
2016.08.26 16:49:10  Parsing configuration from C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml job started
2016.08.26 16:49:10  Reading configuration ...
2016.08.26 16:49:10    Validating XML against XSD ...
2016.08.26 16:49:11    Reading config XML ...
2016.08.26 16:49:11  Collecting exporters ...
2016.08.26 16:49:11    Analyzing exporter 'com.airbus.agi.GEMMA.Exporter.StandardExporter.StandardMappingExporter'
2016.08.26 16:49:11  Parsing configuration from C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml job finished

2016.08.26 16:49:11  Importing ... job started
2016.08.26 16:49:11  Creating database ...
2016.08.26 16:49:11  Importing data ...
2016.08.26 16:49:11    Running parser: com.airbus.agi.GEMMA.Parser.TestParser.TestParserGUIStress
2016.08.26 16:49:22    Running parser: com.airbus.agi.GEMMA.Parser.TestParser.TestParserGUIStress successfully completed.
2016.08.26 16:49:22  Creating the main tree ... job started
2016.08.26 16:49:22  Importing ... job finished

2016.08.26 16:49:22  Rendering ...
2016.08.26 16:49:22    Preparing viewer ...
2016.08.26 16:49:22  Creating tree ...
2016.08.26 16:49:23    Refreshing viewer ...
2016.08.26 16:49:23  Creating the main tree ... job finished

2016.08.26 16:49:23  Reading Rules and Matching ... job started
2016.08.26 16:49:23  Running rules parser.
2016.08.26 16:49:23    Running rules parser...
2016.08.26 16:49:23    Parsing rules file: C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress/Test_Ruleset.xml
2016.08.26 16:49:23    Running rules parser successfully completed.
2016.08.26 16:49:23  Starting Mapper
2016.08.26 16:49:23    Running Mapper: com.airbus.agi.GEMMA.Mapping.GenericMapper
2016.08.26 16:49:23  Preparing Mapping ...
2016.08.26 16:49:29  Mapping ...
2016.08.26 16:53:47  Compacting resource model job started
2016.08.26 16:53:47  Compacting resource model job finished

2016.08.26 16:58:48  Compacting resource model job started
2016.08.26 16:58:51  Compacting resource model job finished
```

(91861 of 102007) Mapping ...

Figure 34 - Old GEMMA Breakdown in GEMMA GUI

Instantly by the multiple comparison of the GEMMA test results and the description made between MapDB results show that the improved MapDB code yields the expected results. As the improved MapDB solution is based on two different set of serialization pattern, the comparison between these two serialization patterns may yield the desired solution for the thesis work in terms of memory footprint reduction. Therefore, considering the improved MapDB test results in 512 MB, 1024 MB, 2048 MB in these three different memory limit which is mentioned in Table 9, 10, 11 respectively for kryo serialization and Table 12, 13, 14 respectively for built-in serialization. The comparison of mapping time (NGMT), resolving time (NGRT) and memory consumption (NGMC) evidently displays that the upgraded MapDB with built-in serialization is better than the MapDB with kryo serialization. The resolving time (NGRT) of MapDB with built-in serialization is approximately 60% - 70% less when compared to resolving time (NGRT) of MapDB kryo serialization and the memory consumption (NGMC) is reduced by 10% - 20 % when compared to the memory consumption (NGMC) of MapDB kryo serialization.

The multiple trials which were performed to check the performance of the MapDB method to achieve the main purpose of the thesis was discussed with the suitable comparisons bring in the conclusion that the upgraded MapDB built-in serialization yields the best results. Ultimately the final comparison of upgraded MapDB built-in serialization result is with the old GEMMA process as shown in Table 12, 13, 14. With respect to the mapping and resolving time consumption the upgraded MapDB solution is 10% - 30% higher when compared to the old Gemma mapping and resolving time. And the memory consumption by each GEMMA mapping process by upgraded MapDB is 10% - 30% less consumed when compared to old GEMMA application.

The main objective which was met after the implementation of MapDB solution was that it was very efficient in the mapping process in any specific memory limit situation, which was not satisfied with the old GEMMA application. As the results of MapDB and old GEMMA comparisons is encountered in Table 12 display that for the huge number of mappables and nodes like example 100000 mappables and nodes are not being able to manage by old GEMMA application as seen in Figure 33 which is recovered by MapDB as seen in Figure 35 with a limited amount of memory consumption without any exceptions. Despite of its slight increase in mapping and resolving time the performance efficiency of MapDB is leading old GEMMA to satisfy the thesis goal to reduce memory footprints of data during runtime. By the entire result of analysis, the MapDB proves to be the solution to store the whole, serialized data on the disk as the database file for depositing mappings, mappable and node data.

Hence the analysis of the chosen solution MapDB to reduce memory footprints during runtime is suitable to achieve the first goal of the thesis work and the second goal to enable storing and restoring of session data is achieved by kryo serialization. By the new GEMMA architecture along with two proposed method MapDB and Kryo serialization each solving the specific goal of the thesis results for the better performance and data efficiency of the GEMMA tool.

The upgraded MapDB solution was very efficent in solving the disadvantage faced by the old GEMMA application. The MapDB completed the whole GEMMA process of 100000 mappable and node data with 512 MB memory limit without any interruption in between. As shown in the Figure 35 which display the memory usage graph of upgraded MapDB in visualvm application.

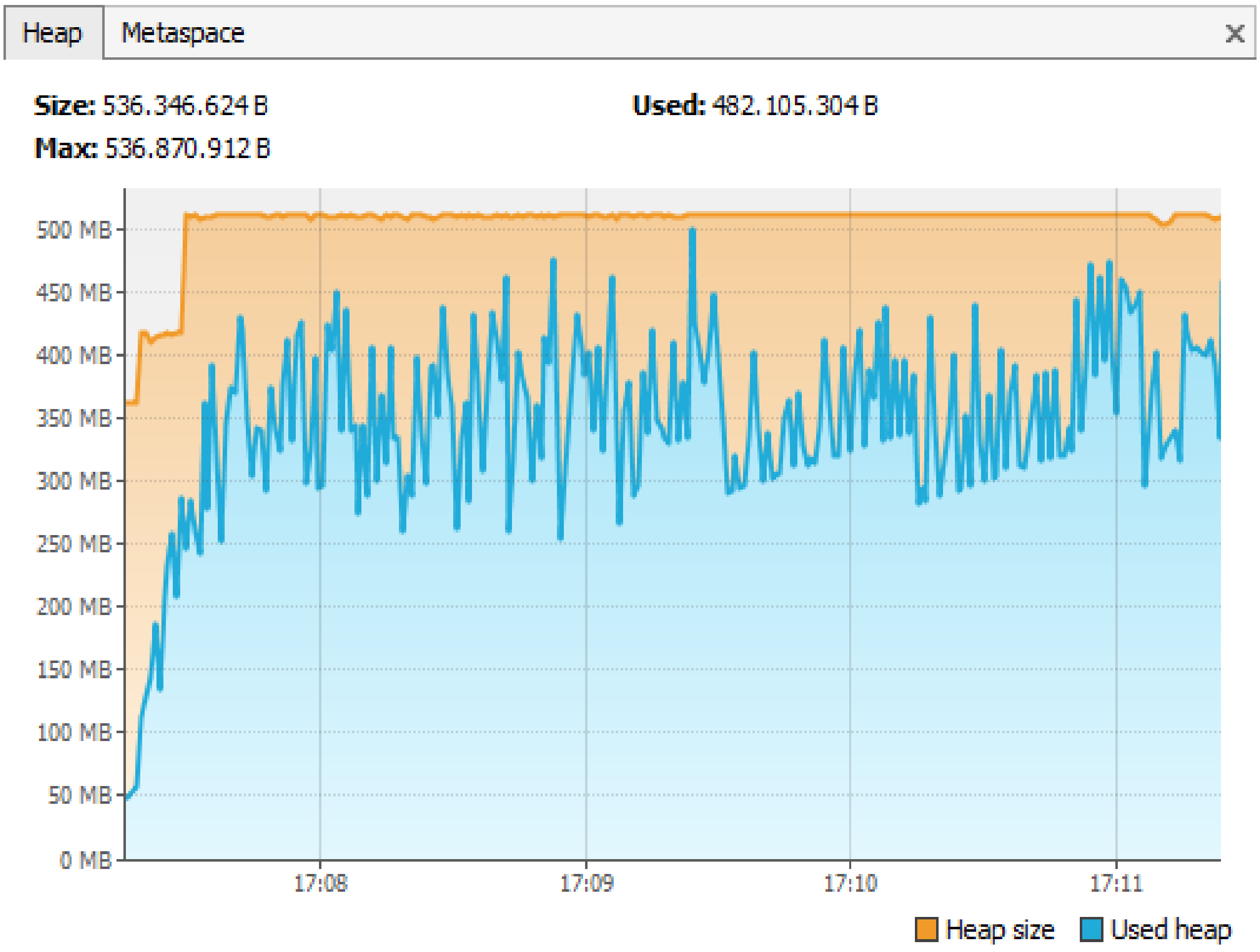
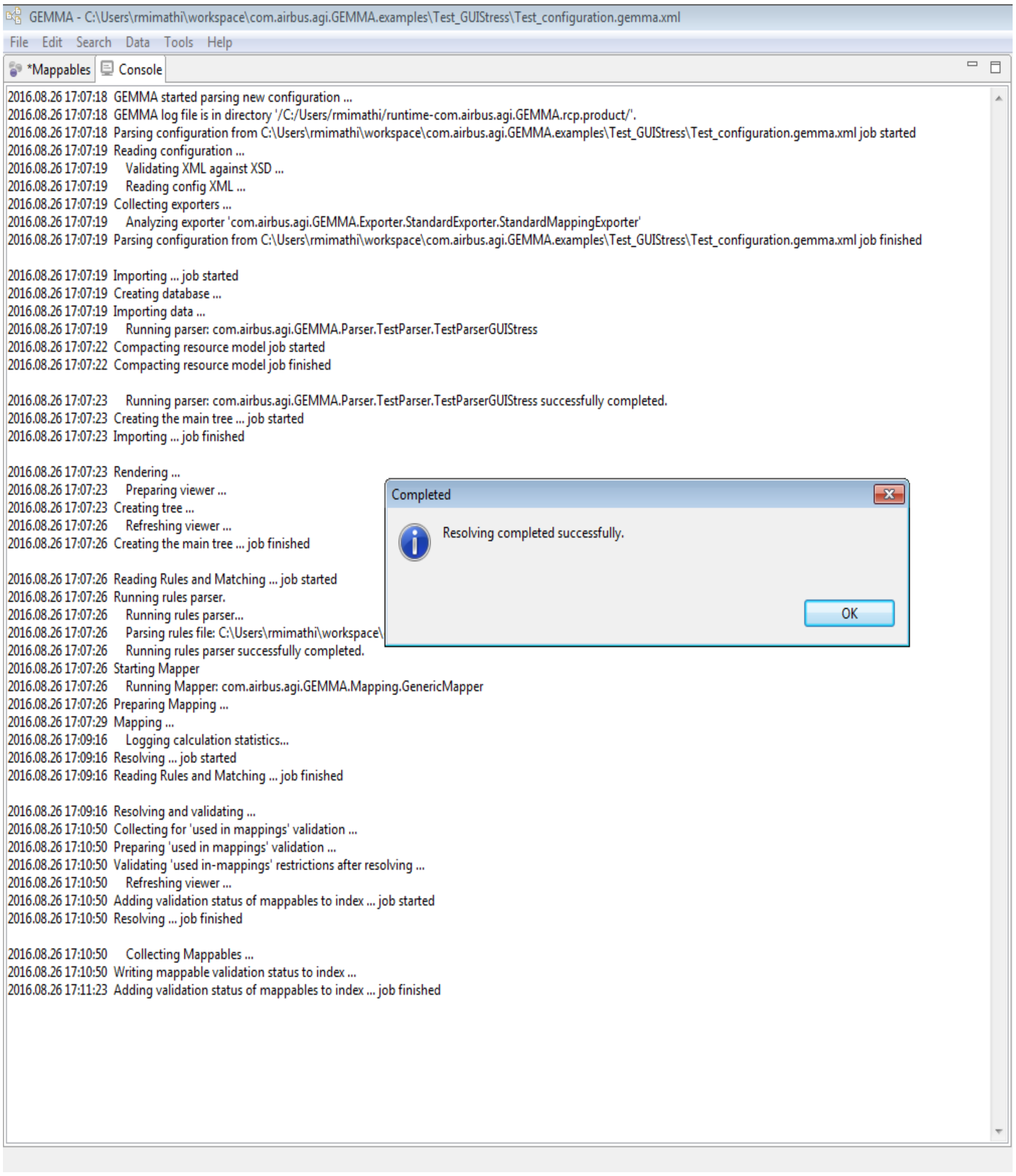


Figure 35 - New GEMMA Process Completion in Java VisualVM

The Figure 36 depicts the upgraded MapDB solution of solving 100000 mappables and nodes data in the GEMMA console tab.


GEMMA - C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml
File Edit Search Data Tools Help
*Mappables
Console
Completed
OK

```
2016.08.26 17:07:18  GEMMA started parsing new configuration ...
2016.08.26 17:07:18  GEMMA log file is in directory '/C:/Users/rmimathi/runtime-com.airbus.agi.GEMMA.rcp.product/'.
2016.08.26 17:07:18  Parsing configuration from C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml job started
2016.08.26 17:07:19  Reading configuration ...
2016.08.26 17:07:19     Validating XML against XSD ...
2016.08.26 17:07:19     Reading config XML ...
2016.08.26 17:07:19  Collecting exporters ...
2016.08.26 17:07:19     Analyzing exporter 'com.airbus.agi.GEMMA.Exporter.StandardExporter.StandardMappingExporter'
2016.08.26 17:07:19  Parsing configuration from C:\Users\rmimathi\workspace\com.airbus.agi.GEMMA.examples\Test_GUIStress\Test_configuration.gemma.xml job finished

2016.08.26 17:07:19  Importing ... job started
2016.08.26 17:07:19  Creating database ...
2016.08.26 17:07:19  Importing data ...
2016.08.26 17:07:19     Running parser: com.airbus.agi.GEMMA.Parser.TestParser.TestParserGUIStress
2016.08.26 17:07:22  Compacting resource model job started
2016.08.26 17:07:22  Compacting resource model job finished

2016.08.26 17:07:23     Running parser: com.airbus.agi.GEMMA.Parser.TestParser.TestParserGUIStress successfully completed.
2016.08.26 17:07:23  Creating the main tree ... job started
2016.08.26 17:07:23  Importing ... job finished

2016.08.26 17:07:23  Rendering ...
2016.08.26 17:07:23     Preparing viewer ...
2016.08.26 17:07:23  Creating tree ...
2016.08.26 17:07:26     Refreshing viewer ...
2016.08.26 17:07:26  Creating the main tree ... job finished

2016.08.26 17:07:26  Reading Rules and Matching ... job started
2016.08.26 17:07:26  Running rules parser.
2016.08.26 17:07:26     Running rules parser...
2016.08.26 17:07:26     Parsing rules file: C:\Users\rmimathi\workspace\
2016.08.26 17:07:26     Running rules parser successfully completed.
2016.08.26 17:07:26  Starting Mapper
2016.08.26 17:07:26     Running Mapper: com.airbus.agi.GEMMA.Mapping.GenericMapper
2016.08.26 17:07:26  Preparing Mapping ...
2016.08.26 17:07:29  Mapping ...
2016.08.26 17:09:16     Logging calculation statistics...
2016.08.26 17:09:16  Resolving ... job started
2016.08.26 17:09:16  Reading Rules and Matching ... job finished

2016.08.26 17:09:16  Resolving and validating ...
2016.08.26 17:10:50  Collecting for 'used in mappings' validation ...
2016.08.26 17:10:50  Preparing 'used in mappings' validation ...
2016.08.26 17:10:50  Validating 'used in-mappings' restrictions after resolving ...
2016.08.26 17:10:50     Refreshing viewer ...
2016.08.26 17:10:50  Adding validation status of mappables to index ... job started
2016.08.26 17:10:50  Resolving ... job finished

2016.08.26 17:10:50     Collecting Mappables ...
2016.08.26 17:10:50  Writing mappable validation status to index ...
2016.08.26 17:11:23  Adding validation status of mappables to index ... job finished
```

Resolving completed successfully.

Figure 36 - New GEMMA Process Completion in GEMMA GUI

5.4 Future Work

During this thesis work, the MapDB and kryo serialization fulfill the objectives and goals which were proposed at the initial stage of the thesis. There is always a wide range of acceptance for the future research and development of the GEMMA framework by utilizing this present thesis results and conclusion.

The results obtained by the thesis work could be analyzed for the future enhancement in the performance of the new approach. The MapDB database engine obtained an acceptable result, while reducing memory footprints during runtime when compared to the old GEMMA framework. During the MapDB operation, it is highly impossible to open the database files which get locked on the disk, if the system crashes without closing the database store. Closing the database store on the disk, plays a vital role in the MapDB framework which leads to a high alert for a GEMMA framework to orderly follow the open and close database store without fail.

By the automatic file lock issue, MapDB could be researched and replaced by a new approach called Xodus according to the results in the reference [30]. Where Xodus is a transactional schema-less embedded high-performance database written in Java. There are few main advantages of Xodus in contrast to MapDB i.e., Xodus provides an outstanding performance due to very compact data storing, lock-free reads, lock-free optimistic writes, and intelligent lock-free caching according to the reference in [31].

In terms of overall system features of the GEMMA, there is always a further contribution made to maintain the GEMMA architecture behavior repeatedly. At present GEMMA is used in most of the research projects in Airbus Defense and Space, but the future aim is to make GEMMA as an open source tool.

Chapter 6

6 Conclusion

The two fundamental goals of the thesis were to reduce the memory footprints of the GEMMA framework during runtime and to enable storing and restoring the session data without re-computing whole data every time. After the thorough analysis of methods and tools, MapDB database engine to reduce memory footprints and kryo serialization to store and restore the session data on the disc, both were considered to achieve the desired goal.

After the selection of two main approach MapDB and kryo, the foremost analysis was on the GEMMA architectural behavior to examine whether the old GEMMA architecture needs any advancement. After the complete analysis of architecture, GEMMA old architecture gave rise to IMappable interface that was implemented by each parser individually. These interfaces gave rise to restriction against data serialization. This lead to the concept of new GEMMA architecture to eliminate the interfaces causing serialization problems.

The new concept of GEMMA new architecture included a new component for centralized data management for the whole project called Data Manager. The centralized Data Manager plays a vital role in managing the huge data and huge data storage on the disk as a database file using MapDB. Later the whole session data could be stored and restored from the disk using kryo serialization.

The new approach implementation of MapDB and kryo was completely tested against the old GEMMA tool with multiple trials of various data collection with various memory limits. The MapDB solution was tested with two different sets of serialization i.e. built serialization of MapDB and kryo serialization. The initial test results of MapDB, proposed for the further advancement in the MapDB code, which was implemented. The final results of the implemented MapDB solution are efficient in reducing memory footprints during runtime as the data is stored on the disk.

MapDB could overcome the old GEMMA drawback by able to solve the issue of GEMMA breakdown while mapping huge amount of data in limited memory consumption E.g. While mapping 100000 mappables and nodes under 512 MB memory limit. The desired results were obtained by the selected approach, MapDB and kryo which was examined, implemented and tested in detail.

7 Appendix A: MapDB Using Built-in Serialization

```
    package com.airbus.agi.GEMMA.DataManagement;

import java.io.File;
import java.io.Serializable;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.mapdb.DB;
import org.mapdb.DBMaker;
import org.mapdb.HTreeMap;
import org.mapdb.Serializer;

import com.airbus.agi.GEMMA.Mapping.Mappable;
import com.airbus.agi.GEMMA.Mapping.Mapping;
import com.airbus.agi.GEMMA.Mapping.Node;

public class DataManager implements Serializable {

    private static final long serialVersionUID = -4069122550495907255L;

    private static String cacheFolderAbsolutePath = null;

    final static String CACHE_FOLDER_NAME = "cache";

    //Get actual class name to be printed on
    static Logger log = LogManager.getLogger(DataManager.class.getName());

    //create an object of SingleObject
    private static DataManager instance = new DataManager();

    //Get the singleton
    public static DataManager getInstance(){
        return instance;
    }

    public static void setInstance(DataManager manager) {
        instance = manager;
    }

    private transient  DB dbMappableDisk;
    private transient  HTreeMap onMappableDisk;

    private transient DB dbNodeDisk;
    private transient  HTreeMap onNodeDisk;

    private transient DB dbMappingDisk;
    private transient  HTreeMap onMappingDisk;


    public void initialize() {

        dbMappableDisk  = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/mappable.db"))
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappableDisk = (HTreeMap<String,
Object>)dbMappableDisk.hashMap("onMappableDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbNodeDisk = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/node.db"))
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onNodeDisk = (HTreeMap<String,
Object>)dbNodeDisk.hashMap("onNodeDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbMappingDisk = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/mappings.db"))
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappingDisk = (HTreeMap<String,
Object>)dbMappingDisk.hashMap("onMappingDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

    }

    public void reOpenOldDatabaseFiles() {

        dbMappableDisk  = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/mappable.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappableDisk = (HTreeMap<String,
Object>)dbMappableDisk.hashMap("onMappableDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbNodeDisk = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/node.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onNodeDisk = (HTreeMap<String,
Object>)dbNodeDisk.hashMap("onNodeDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbMappingDisk = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/mappings.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappingDisk = (HTreeMap<String,
Object>)dbMappingDisk.hashMap("onMappingDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        }


    public void shutdown() {
        if (dbMappableDisk != null) {
            dbMappableDisk.commit();
            dbMappableDisk.close();
        }

        if (dbNodeDisk != null) {
            dbNodeDisk.commit();
            dbNodeDisk.close();
        }

        if (dbMappingDisk != null) {
            dbMappingDisk.commit();
            dbMappingDisk.close();
        }

    }

    public void close1() {
        if (dbMappableDisk != null) {
            dbMappableDisk.close();
        }

        if (dbNodeDisk != null) {
            dbNodeDisk.close();
        }

        if (dbMappingDisk != null) {
            dbMappingDisk.close();
        }

    }

    public void clearMappings() {

        onMappingDisk.clear();
    }
    public void addMappable(Mappable mappable) {

        HashMap<String, Mappable> mappables = new HashMap<String,
Mappable>();
        onMappableDisk.put(mappable.getID(), mappable);
        addMappables(mappables);
    }

    public void addMappables(HashMap<String, Mappable> mappables) {
        for (Mappable mappable : mappables.values()) {
            addMappable(mappable);
        }
        if(mappables.size() != 0){
        log.debug("Adding Mappables to Lucene Database");
        MappableDatabase.getInstance().addMappablesToIndex(mappables);
        }
    }

    public void updateMappable(Mappable mappable) {
        this.onMappableDisk.put(mappable.getID(), mappable);
        MappableDatabase.getInstance().updateMappable(mappable);
    }

    public void deleteMappable(String mappableID) {

        onMappableDisk.remove(mappableID);
        MappableDatabase.getInstance().deleteMappable(mappableID);
    }

    public void deleteMappables(HashSet<String> mappableIDs) {
        for (String mappableID : mappableIDs) {
            deleteMappable(mappableID);
        }
    }

    public Set<String> getAllMappableKeys() {
        return onMappableDisk.keySet();
    }

    public HashMap<String, Mappable> getAllMappables() {

        HashMap<String, Mappable> mappables = new HashMap<String,
Mappable>();
        for (Object key : onMappableDisk.keySet()) {
            mappables.put(((Mappable)onMappableDisk.get(key)).getID(),
(Mappable)onMappableDisk.get(key));
        }
        return mappables;
    }

    public Mappable getMappable(String mappableID) {

        return ((Mappable)onMappableDisk.get(mappableID));
    }

    public Set<String> getAllNodeKeys() {
        return onNodeDisk.keySet();
    }

    public Node getNode(String nodeID) {
        return ((Node)onNodeDisk.get(nodeID));
    }

    public void removeNode(String nodesID) {
        onNodeDisk.remove(nodesID);
        MappableDatabase.getInstance().deleteMappable(nodesID);
    }

    public void removeNodes(HashSet<String> nodesIDs) {
        for (String id : nodesIDs) {
            onNodeDisk.remove(id);
        }
    }

    @SuppressWarnings("unchecked")
    public void addNode(Node node) {
        HashMap<String, Node> nodes = new HashMap<String, Node>();
        onNodeDisk.put(node.getID(), node);
        addNodes(nodes);
    }

    public void addNodes(HashMap<String, Node> nodes) {
        for (Node node : nodes.values()) {
            addNode(node);
        }

    }

    public void addMapping(Mapping mapping) {
        onMappingDisk.put(mapping.getFromMappable().getID(), mapping);
    }

    public void addMappings(HashMap<String, Mapping> mappings) {
        for (Mapping mapping : mappings.values()) {
            addMapping(mapping);
        }

    }

    public void deleteMapping(String fromMappableID) {
        onMappingDisk.remove(fromMappableID);
        MappableDatabase.getInstance().deleteMappable(fromMappableID);
    }

    public void deleteMappings(HashSet<String> fromMappableIDs) {
        for (String fromMappableID : fromMappableIDs) {
            onMappingDisk.remove(fromMappableID);
        }
    }

    public Set<String> getAllMappingKeys() {
        return onMappingDisk.keySet();
    }

    public Mapping getMapping(String mappingID) {
        return ((Mapping)onMappingDisk.get(mappingID));
    }

    private Object readResolve() {
        return instance;
    }
    public boolean fillDatabaseAfterRestore() {
        boolean success = true;
        log.debug("Adding Mappables to Lucene Database");
        success = success &&
MappableDatabase.getInstance().addMappablesToIndex(this.getAllMappables());
        return success;
    }


    public static String getCacheFolderAbsolutePath() {
        return cacheFolderAbsolutePath;
    }

    public static void setCacheFolderAbsolutePath(String
cacheFolderAbsolutePath) {
        DataManager.cacheFolderAbsolutePath = cacheFolderAbsolutePath;
    }

}
```

8 Appendix B: Kryo Serialization

```
package com.airbus.agi.GEMMA.Util;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import org.apache.commons.lang3.exception.ExceptionUtils;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.osgi.framework.Bundle;
import org.osgi.framework.FrameworkUtil;
import org.osgi.framework.wiring.BundleWiring;

import com.airbus.agi.GEMMA.Configuration.GEMMA_Parameters;
import com.airbus.agi.GEMMA.DataManagement.DataManager;
import com.airbus.agi.GEMMA.GUI.utls.GUIUtls;
import com.airbus.agi.GEMMA.Resolver.IResolver;
import com.esotericsoftware.kryo.Kryo;
import com.esotericsoftware.kryo.io.Input;
import com.esotericsoftware.kryo.io.Output;
import com.esotericsoftware.kryo.util.MapReferenceResolver;

import net.lingala.zip4j.core.ZipFile;
import net.lingala.zip4j.exception.ZipException;
import net.lingala.zip4j.model.FileHeader;
import net.lingala.zip4j.model.ZipParameters;
import net.lingala.zip4j.util.Zip4jConstants;

public class GEMMASerializeUtls {

    static Logger log =
LogManager.getLogger(GEMMASerializeUtls.class.getName());

    public static Object deserialize(String filePath, Object objectToRead,
IProgressMonitor monitor){

        ExtClassResolver cr = new ExtClassResolver();
        Kryo kryo = new Kryo(cr, new MapReferenceResolver());

        // Resolver may be in a different plugin. We need to find the
correct class loader
        if (objectToRead instanceof IResolver) {
            Bundle bundle =
FrameworkUtil.getBundle(objectToRead.getClass());
            BundleWiring bundleWiring = bundle.adapt(BundleWiring.class);
            ClassLoader classLoader = bundleWiring.getClassLoader();
            kryo.setClassLoader(classLoader);
        }
        kryo.register(objectToRead.getClass(),
objectToRead.getClass().hashCode());
```

```
        Input input;
        Object restoredObject = null;
        String msg;
        try {
            input = new Input(new FileInputStream(filePath));
            try {
                try {
                    restoredObject = kryo.readObject(input,
objectToRead.getClass());
                    msg = "Restored successfully from '" + filePath + "'";
                    monitor.setTaskName(msg);
                    log.debug(msg);
                    input.close();
                } catch (Exception e) {
                    msg = "Restoring failed from '" + filePath + "'.
Deserialization error, see log.";
                    monitor.setTaskName(msg);
                    input.close();
                    log.error(msg);
                    log.error(ExceptionUtils.getStackTrace(e));
                }
            } catch (Exception e) {
                msg = "Restoring failed from '" + filePath + "'. See log.
Will need to re-compute data.";
                monitor.setTaskName(msg);
                input.close();
                log.error(msg);
                log.error(ExceptionUtils.getStackTrace(e));
            }
        } catch (FileNotFoundException e) {
            msg = "Restoring failed from '" + filePath + "'. The file was
not found. Will need to re-compute data.";
            monitor.setTaskName(msg);
            log.error(msg);
            log.error(ExceptionUtils.getStackTrace(e));
        }

        return restoredObject;
    }

    public static boolean serialize(String filePath, Object
objectToSerialize, IProgressMonitor monitor){
        boolean isSuccess = true;
        ExtClassResolver cr = new ExtClassResolver();
        Kryo kryo = new Kryo(cr, new MapReferenceResolver());
        kryo.register(objectToSerialize.getClass(),
objectToSerialize.getClass().hashCode());

        try {
            Output output = new Output(new FileOutputStream(filePath));
            try {
                kryo.writeObject(output, objectToSerialize);
                String msg = "Storing "+filePath + "";
                monitor.setTaskName(msg);
                log.debug(msg);
                output.flush();
                output.close();
            } catch (Exception e) {
                String msg = "Storing failed for "+filePath + ".
Incompatible format.";
                monitor.setTaskName(msg);
                isSuccess = false;
                output.flush();
                output.close();
                log.error(msg);
                log.error(ExceptionUtils.getStackTrace(e));
            }
            output.flush();
            output.close();
        } catch (FileNotFoundException e) {
            String msg = "Storing failed for "+filePath + ". File not
found.";
            monitor.setTaskName(msg);
            isSuccess = false;
            log.error(msg);
            log.error(ExceptionUtils.getStackTrace(e));
            }
        return isSuccess;
    }

    public static boolean zip(String zipFileAbsolutePath, ArrayList<String>
filesPaths, IProgressMonitor monitor){
        try {
            HashSet<File> tempFilesToDelete = new HashSet<File>();
            // delete old file.
            new File(zipFileAbsolutePath).delete();

            ZipFile zipFile = new ZipFile(zipFileAbsolutePath);

            ZipParameters parameters = new ZipParameters();
            parameters.setCompressionMethod(Zip4jConstants.COMP_DEFLATE);
            // set compression method to deflate compression
            // Set the compression level. This value has to be in between 0
to 9
            // Several predefined compression levels are available
            // DEFLATE_LEVEL_FASTEST - Lowest compression level but higher
speed of compression
            // DEFLATE_LEVEL_FAST - Low compression level but higher speed
of compression
            // DEFLATE_LEVEL_NORMAL - Optimal balance between compression
level/speed
            // DEFLATE_LEVEL_MAXIMUM - High compression level with a
compromise of speed
            // DEFLATE_LEVEL_ULTRA - Highest compression level but low
speed

parameters.setCompressionLevel(Zip4jConstants.DEFLATE_LEVEL_NORMAL);

            ArrayList<File> dataFiles = new ArrayList<File>();
            ArrayList<File> cacheFiles = new ArrayList<File>();
            ArrayList<File> configFiles = new ArrayList<File>();

            for (String filePath : filesPaths) {
                if
(filePath.endsWith(GEMMAConstants.GEMMA_SERIALIZED_FILE_EXTENSION)) {
                    dataFiles.add(new File(filePath));
                }
                else if
(filePath.endsWith(GEMMAConstants.GEMMA_CACHE_FILE_EXTENSION)) {
                    cacheFiles.add(new File(filePath));
                }
                else if
(filePath.endsWith(GEMMAConstants.GEMMA_CONFIGURATION_FILE_EXTENSION)) {
```

```
                File config = new File(filePath);
                configFiles.add(config);
            }
        }

        zipFile.addFiles(configFiles, parameters);


parameters.setRootFolderInZip(GEMMA_Parameters.GEMMA_CACHE_DIR_NAME);
        zipFile.addFiles(dataFiles, parameters);
        zipFile.addFiles(cacheFiles, parameters);

        String msg = "Compressed successfully.";
        log.debug(msg);
        GUIUtls.setMonitorTaskName(monitor, msg);

        // clean up
        msg = "Cleaning up, deleting files used temporary.";
        log.debug(msg);
        GUIUtls.setMonitorTaskName(monitor, msg);

        tempFilesToDelete.addAll(dataFiles);
        // clean up
        for (File file : tempFilesToDelete) {
            file.delete();
        }

        // delete empty cache dir
        File cacheDir = new
File(FileUtil.getDirectoryPath(zipFileAbsolutePath) + "/" +
GEMMA_Parameters.GEMMA_CACHE_DIR_NAME);
        if (cacheDir.exists()) {
            cacheDir.delete();
        }

        return true;

    } catch (ZipException e) {
        String msg = "Compression failed. See log.";
        log.error(msg);
        GUIUtls.setMonitorTaskName(monitor, msg);
        log.error(msg);
        log.error(ExceptionUtils.getStackTrace(e));
        return false;
    }
}

@SuppressWarnings("rawtypes")
public static ArrayList<String> unzip(String zipFileAbsolutePath,
String projectDirAbsolutePath, IProgressMonitor monitor, Shell
applicationShell){

    try {
        ArrayList<String> extractedFilesNames = new
ArrayList<String>();

        ZipFile zipFile = new ZipFile(zipFileAbsolutePath);

        // Get the list of file headers from the zip file
        List fileHeaderList = zipFile.getFileHeaders();

        for (int i = 0; i < fileHeaderList.size(); i++) {
                FileHeader fileHeader = (FileHeader)fileHeaderList.get(i);
                if
(fileHeader.getFileName().endsWith(GEMMAConstants.GEMMA_CONFIGURATION_FILE_
EXTENSION)) {
                    File configFile = new File(projectDirAbsolutePath + "/"
+ fileHeader.getFileName());
                    if (configFile.exists()) {
                        Display.getDefault().syncExec(new Runnable() {

                            public void run() {
                                boolean override =
MessageDialog.openQuestion(applicationShell, "Confirm", "Overwrite " +
configFile + "?");
                                if (!override) {

fileHeader.setFileName(GEMMAConstants.GEMMA_CONFIGURATION_TEMP_FILE_PREFIX
+ fileHeader.getFileName());
                                    log.info("Using existing configuration
file" + configFile);
                                }
                                else {
                                    log.info("Overwriting the existing file
with the one from .gma archive " + configFile);

                                }
                            }
                        });
                    }
                }
                extractedFilesNames.add(fileHeader.getFileName());

            }
//          DataManager.getInstance().shutdown();
            zipFile.extractAll(projectDirAbsolutePath);

            return extractedFilesNames;

        } catch (ZipException e) {
            log.error("Decompression failed. See log.");
            log.error(ExceptionUtils.getStackTrace(e));
        }
        return null;
    }

}
```

9 Appendix C: MapDB Using Kryo Serialization

```java
package com.airbus.agi.GEMMA.DataManagement;

import java.io.File;
import java.io.Serializable;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.mapdb.DB;
import org.mapdb.DBMaker;
import org.mapdb.HTreeMap;
import org.mapdb.Serializer;
import com.airbus.agi.GEMMA.DataManagementTest.SerializationWrapper;
import com.airbus.agi.GEMMA.Mapping.Mappable;
import com.airbus.agi.GEMMA.Mapping.Mapping;
import com.airbus.agi.GEMMA.Mapping.Node;

public class DataManager  implements Serializable {

    private static final long serialVersionUID = -4069122550495907255L;

    private static String cacheFolderAbsolutePath = null;

    final static String CACHE_FOLDER_NAME = "cache";

    //Get actual class name to be printed on
    static Logger log = LogManager.getLogger(DataManager.class.getName());

    //create an object of SingleObject
    private static DataManager  instance = new DataManager ();

    //private constructor
    private DataManager  () {

    }

    //Get the singleton
    public static DataManager  getInstance(){
        return instance;
    }

    public static void setInstance(DataManager manager) {
        instance = manager;
    }

    private transient  DB dbMappableDisk;
    private transient  HTreeMap onMappableDisk;

    private transient DB dbNodeDisk;
    private transient  HTreeMap onNodeDisk;

    private transient DB dbMappingDisk;
    private transient  HTreeMap onMappingDisk;

    @SuppressWarnings("unchecked")
    public void initialize() {
        dbMappableDisk  = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/mappable.db"))
                .fileMmapEnable()
                .fileMmapEnableIfSupported()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappableDisk = (HTreeMap<String,
Object>)dbMappableDisk.hashMap("onMappableDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbNodeDisk = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/node.db"))
                .fileMmapEnable()
                .fileMmapEnableIfSupported()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onNodeDisk = (HTreeMap<String,
Object>)dbNodeDisk.hashMap("onNodeDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbMappingDisk = DBMaker
                .fileDB(new File(cacheFolderAbsolutePath + "/mappings.db"))
                .fileMmapEnable()
                .fileMmapEnableIfSupported()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappingDisk = (HTreeMap<String,
Object>)dbMappingDisk.hashMap("onMappingDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

    }

    @SuppressWarnings("unchecked")
    public void reOpenOldDatabaseFiles() {

        dbMappableDisk  = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/mappable.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappableDisk = (HTreeMap<String,
Object>)dbMappableDisk.hashMap("onMappableDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbNodeDisk = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/node.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onNodeDisk = (HTreeMap<String,
Object>)dbNodeDisk.hashMap("onNodeDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();

        dbMappingDisk = DBMaker
                .fileDB(cacheFolderAbsolutePath + "/mappings.db")
                .fileMmapEnable()
                .fileMmapPreclearDisable()
                .cleanerHackEnable()
                .closeOnJvmShutdown()
                .make();

        onMappingDisk = (HTreeMap<String,
Object>)dbMappingDisk.hashMap("onMappingDisk")
                .keySerializer(Serializer.STRING)
                .hashSeed(111)
                .createOrOpen();
        }


    public void shutdown() {
        if (dbMappableDisk != null) {
            dbMappableDisk.commit();
            dbMappableDisk.close();
        }

        if (dbNodeDisk != null) {
            dbNodeDisk.commit();
            dbNodeDisk.close();
        }

        if (dbMappingDisk != null) {
            dbMappingDisk.commit();
            dbMappingDisk.close();
        }

    }

    public void close1() {
        if (dbMappableDisk != null) {

            dbMappableDisk.close();
        }

        if (dbNodeDisk != null) {

            dbNodeDisk.close();
        }
        if (dbMappingDisk != null) {

            dbMappingDisk.close();
        }

    }


    public void clearMappings() {

        onMappingDisk.clear();
    }

    @SuppressWarnings("unchecked")
    public void addMappable(Mappable mappable) {

        HashMap<String, Mappable> mappables = new HashMap<String,
Mappable>();
        onMappableDisk.put(mappable.getID(), new
SerializationWrapper(mappable));
        addMappables(mappables);
    }

    public void addMappables(HashMap<String, Mappable> mappables) {
        for (Mappable mappable : mappables.values()) {
            addMappable(mappable);
        }

        if(mappables.size() != 0){
        log.debug("Adding Mappables to Lucene Database");
        MappableDatabase.getInstance().addMappablesToIndex(mappables);
        }
    }

    public void updateMappable(Mappable mappable) {
        this.onMappableDisk.put(mappable.getID(), new
SerializationWrapper(mappable));
        MappableDatabase.getInstance().updateMappable(mappable);
    }

    public void deleteMappable(String mappableID) {
        onMappableDisk.remove(mappableID);
        MappableDatabase.getInstance().deleteMappable(mappableID);
    }

    public void deleteMappables(HashSet<String> mappableIDs) {
        for (String mappableID : mappableIDs) {
            deleteMappable(mappableID);
        }
    }

    public HashMap<String, Mappable> getAllMappables() {

        HashMap<String, Mappable> mappables = new HashMap<String,
Mappable>();
        for (Object key : onMappableDisk.keySet()) {
            SerializationWrapper wrapper =
(SerializationWrapper)onMappableDisk.get(key);
            mappables.put((((Mappable)wrapper.getWrappedObject())).getID(),
((Mappable)wrapper.getWrappedObject()));
        }
        return mappables;
    }

    public Set<String> getAllMappableKeys() {
        return onMappableDisk.keySet();
    }

    public Mappable getMappable(String mappableID) {
        SerializationWrapper wrapper =
(SerializationWrapper)onMappableDisk.get(mappableID);
        if (wrapper != null) {
            return (Mappable)wrapper.getWrappedObject();
        } else {
            return null;
        }

    }

    public Set<String> getAllNodeKeys() {
        return onNodeDisk.keySet();
    }

    public Node getNode(String nodeID) {
        SerializationWrapper wrapper =
(SerializationWrapper)onNodeDisk.get(nodeID);
        if (wrapper != null) {
            return (Node)wrapper.getWrappedObject();
        } else {
            return null;
        }

    }

    public void removeNode(String nodesID) {
        onNodeDisk.remove(nodesID);
        MappableDatabase.getInstance().deleteMappable(nodesID);
    }

    public void removeNodes(HashSet<String> nodesIDs) {
        for (String id : nodesIDs) {
            onNodeDisk.remove(id);
        }
    }

    @SuppressWarnings("unchecked")
    public void addNode(Node node) {
        HashMap<String, Node> nodes = new HashMap<String, Node>();
        onNodeDisk.put(node.getID(), new SerializationWrapper(node));
        addNodes(nodes);
    }

    public void addNodes(HashMap<String, Node> nodes) {
        for (Node node : nodes.values()) {
            addNode(node);
        }
    }

    public void addMapping(Mapping mapping) {
        onMappingDisk.put(mapping.getFromMappable().getID(), new
SerializationWrapper(mapping));
    }
    public void addMappings(HashMap<String, Mapping> mappings) {
        for (Mapping mapping : mappings.values()) {
            addMapping(mapping);
        }

    }

    public void deleteMapping(String fromMappableID) {
        onMappingDisk.remove(fromMappableID);
        MappableDatabase.getInstance().deleteMappable(fromMappableID);
    }

    public void deleteMappings(HashSet<String> fromMappableIDs) {
        for (String fromMappableID : fromMappableIDs) {
            onMappingDisk.remove(fromMappableID);
        }
    }

    public Set<String> getAllMappingKeys() {
        return onMappingDisk.keySet();
    }

    public Mapping getMapping(String mappingID) {
        SerializationWrapper wrapper =
(SerializationWrapper)onMappingDisk.get(mappingID);
        if (wrapper != null) {
            return (Mapping)wrapper.getWrappedObject();
        } else {
            return null;
        }
    }

    private Object readResolve() {
        return instance;
    }

    public boolean fillDatabaseAfterRestore() {
        boolean success = true;
        log.debug("Adding Mappables to Lucene Database");
        success = success &&
MappableDatabase.getInstance().addMappablesToIndex(this.getAllMappables());
        return success;
    }


    public static String getCacheFolderAbsolutePath() {
        return cacheFolderAbsolutePath;
    }

    public static void setCacheFolderAbsolutePath(String
cacheFolderAbsolutePath) {
        DataManager .cacheFolderAbsolutePath = cacheFolderAbsolutePath;
    }

}
```

10 References

[1] Philipp Helle and Wladimir Schamai, "Using a Generic Modular Mapping Framework for Simulation Model Composition", 2015 7th International Conference on Advances in System Simulation
[2] (5 August, 2016). Oracle documentation. Interface Serializable. [Online].
Available: https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
[3] Serialization and deserialization in Java. [Online]. **Available:** http://www.codingeek.com/wp-content/uploads/2014/11/Serialization-deserialization-in-Java-Object-Streams.jpg
[4] Margo. (2015). Kryo 3.0.0. [Online] **Available:** https://github.com/EsotericSoftware/kryo
[5] James Sutherland. (2013). Java Persistence Performance. [Online]. **Available:** http://java-persistence-performance.blogspot.de/2013/08/optimizing-java-serialization-java-vs.html
[6] Hibernate Overview. [Online].
Available: http://www.tutorialspoint.com/hibernate/hibernate_overview.htm
[7] (07 July 2016). Hibernate (framework). [Online].
Available: https://en.wikipedia.org/wiki/Hibernate_(framework)
[8] (2009). Intro to Caching, Caching algorithm and caching frameworks part 1. [Online].
Available: http://javalandscape.blogspot.de/2009/01/cachingcaching-algorithms-and-caching.html
[9] (2015). About Ehcache. [Online].
Available: http://www.ehcache.org/generated/2.10.2/html/ehc-all/#page/Ehcache_Documentation_Set%2Fto-title_about_ehcache.html%23
[10] (07 October, 2016). JCS vs EHCache Memory Performance. [Online]. **Available:** https://commons.apache.org/proper/commons-jcs/JCSvsEHCache.html
[11] (August 3 2015). Analysis of JVM off-Heap Caching libraries. [Online]. **Available:** http://engineering.snapdeal.com/analysis-of-jvm-off-heap-caching-libraries-201508/
[12] Joshua Bloch, "Serialization", in Effective Java, 2nd ed. Addison-Wesley Professional, 2008
[13] (09 September 2016). MapDB: Database engine [Online].
Available: https://github.com/jankotek/mapdb
[14] Lucy Carey. (May 16, 2014). MapDB which is a pure Java database, for Java developers. [Online]. **Available:** https://jaxenter.com/cory-isaacson-mapdb-is-a-pure-java-database-for-java-developers-107799.html
[15] Cristophe. (May 13, 2016). How does a HashMap work in Java. [Online]. **Available:** http://coding-geek.com/how-does-a-hashmap-work-in-java/
[16] (08 June, 2016) Oracle documentation. Class HashSet<E>. [Online]. **Available:** https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html
[17] A. Mitschke et al., "CRYSTAL public aerospace use case Development Report - V2," ARTEMIS EU CRYSTAL project, Tech. Rep. D208.902, 2015.
[18] T. Blochwitz et al., "The functional mockup interface for tool independent exchange of simulation models," in 8th International Modelica Conference, Dresden, 2011, pp. 20–22.

[19] (23 October 2016). MapDB Introduction. [Online]
Available: https://jankotek.gitbooks.io/mapdb/content/
[20] (02 October 2016). HTreeMap. [Online].
Available: https://jankotek.gitbooks.io/mapdb/content/htreemap/
[21] (21 September 2016). Performance durability. [Online].
Available: https://jankotek.gitbooks.io/mapdb/content/performance/
[22] (15 September 2016). Class DBMaker.Maker. [Online]. **Available:** http://www.mapdb.org/javadoc/latest/mapdb/org/mapdb/DBMaker.Maker.html#fileMmapPreclearDisable--
[23] (06 September 2016) Who is using MapDB. [Online]. **Available:** http://www.mapdb.org/success/
[24] Tiago Marques Godinho, "DISTRIBUTED PACS: PERFORMANCE AND AVAILABILITY," M.S thesis, Dept. Electronic Telecommunication and Informatik, University de Aveiro, 2013.
[25] (2016) MinLog. [Online]. **Available:** https://github.com/EsotericSoftware/minlog/
[26] (2016) Objenesis. [Online]. **Available:** https://github.com/easymock/objenesis
[27] (2016) ReflectASM. [Online]. **Available:**https://github.com/EsotericSoftware/reflectasm
[28] Ahmed Ali. (March 24, 2009). Intro to Caching, Caching algorithms and caching frameworks part 4. [Online]. **Available:** http://javalandscape.blogspot.de/2009/03/intro-to-cachingcaching-algorithms-and.html
[29] Jiri Sedlacek and Tomas Hurka. (2016). VisualVM. [Online]. **Available:** https://visualvm.github.io/features.html
[30] (2016) Xodus. [Online]. **Available:** http://jetbrains.github.io/xodus/#download
[31] (2016) Xodus Overview. [Online]. **Available:** https://github.com/JetBrains/xodus/wiki
[32] Concepts Related to Caching. [Online]. **Available:** http://www.ehcache.org/documentation/3.1/caching-concepts.html
[33] (07 October, 2016). Java Caching System. [Online]. **Available:** https://commons.apache.org/proper/commons-jcs/