|  |
| --- |
| TU Wien – Institut für Computertechnik |
| Gardener – Java to plantUML parser |
| Software Design Description (SDD) |

|  |  |  |  |
| --- | --- | --- | --- |
| Gruppe | Autor(en) | | MatrikelNr. |
| 5 | Krevastouli Eirini | | 11845882 |
| Telleria Manuel | | 01429517 |
| Ausgabe-Datum: 16.12.2021 | | Version: 1.0 | |

# Table of contentS

[0 Table of contentS 1](#_Toc94201422)

[0.1 Version History 1](#_Toc94201423)

[1 Introduction 2](#_Toc94201424)

[2 System Description 2](#_Toc94201425)

[2.1 External Interfaces of the system 3](#_Toc94201426)

[2.2 Architectural Description 3](#_Toc94201427)

[3 USER INTERFACE LAYER DESCRIPTION 5](#_Toc94201428)

[3.1 Interface 5](#_Toc94201429)

[3.2 Structural description 6](#_Toc94201430)

[3.3 Behavioral description 7](#_Toc94201431)

[4 BUSINESS OBJECT LAYER DESCRIPTION 7](#_Toc94201432)

[4.1 Interface 8](#_Toc94201433)

[4.2 Structural description 10](#_Toc94201434)

[4.3 Behavioral description 11](#_Toc94201435)

[5 DATA LAYER DESCRIPTION 12](#_Toc94201436)

[5.1 Interface 12](#_Toc94201437)

[5.2 Structural description 13](#_Toc94201438)

[5.3 Behavioral description 14](#_Toc94201439)

[6 External references 15](#_Toc94201440)

[7 Annex 15](#_Toc94201441)

## Version History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Release notes** |
| 0.1.0 | 2021-11-20 | Creation of the Document |
| 0.1.1 | 2021-12-15 | First working draft |
| 1.0 | 2021-27-01 | Final Revision |

Tabelle 1: Version History

# Introduction

The PlantUML language provides an easy way for developers to quickly generate UML diagrams from text files. The goal of this project is to further simplify this process by providing a java software library to automatically generate UML sequence diagrams from the source code of valid java (1.8 extension) methods.

The purpose of this document is to precisely describe the capabilities, functionality, and structure of the Gardener library to provide stakeholders with a working understanding of the project and help facilitate the operation of the Gardener software library by any stakeholders interested in automatically creating plantUML sequence diagrams from the source code of valid java (1.8 extension) methods.

To accomplish this, a precise description of the total system as well as every contained subsystem and how each subsystem relates to the others is provided inside this document.

This document also deals with the general description of the Project software architecture and the description of the specific developed system.

The document is intended for both the stakeholders and the developers of the Gardener library and will be proposed to the primary stakeholders for approval.

# System Description

The developed system can be broadly divided into three component subsystems.

The parsing subsystem, responsible for the extraction of information from the java source code provided by the user of the library, uses a combination of the java built in reflection API and the parsing of an abstract syntax tree generated by the JavaParser library ([JavaParser - Home](https://javaparser.org/)) to extract as much information as necessary to create the appropriate plantUML sequence diagram from the java source code provided by the user.

The input/output subsystem is responsible for storing the information obtained by the parsing subsystem in a format that allows the creation of the necessary plantUML syntax text file needed by the plantuml software library to generate the appropriate sequence diagram image. This subsystem is also responsible for generating the plantUML syntax text file from the stored information using the java built in input/output API.

The user interface subsystem is responsible for letting the user of the library configure the library to achieve the desired results when generating the plantUML sequence diagram. It is also responsible for managing the other subsystems and passing on the appropriate information as provided by the user of the library to those parts of the system where the information is needed.

A messaging system based around events is also used to communicate information between different subsystems.

## External Interfaces of the system

The external interfaces of the system the user needs to interact with are totally contained in the user interface subsystem.

The main goal of the gardener library is to reduce the effort required for creating plantUML sequence diagrams of a given java method. To accomplish this goal, a lot of effort was put into minimizing the required user input at all stages of the process.

A user interacts with the library by simply instantiating a diagram generator object and specifying the desired parsing behavior. The user can then request the diagram generator object to create a sequence diagram from a specific java source code file.

The minimum amount of information required to carry out this operation must also be passed to the diagram generator object. This information includes:

1. A file path to the file containing the source code of the method the user wishes to make a diagram of as a java string
2. A file path to the location the user wants the final sequence diagram syntax file and image to be stored in as a java string
3. The name of the method the user wants to generate a sequence diagram of as a java string
4. The instantiated class that defines the method the user wants to generate a sequence diagram as a java object
5. The name of the class that defines the method the user wants to generate a sequence diagram of as a java string
6. The name of the package that contains the class that defines the method the user wants to generate a sequence diagram of as a java string
7. A collection of java class objects that correspond to the types of every argument to the method the user wants to generate a sequence diagram of as a java vararg

To remain as flexible as possible, the specifics of how a user instantiates and interacts with the diagram generator object as well as the specifics of how a user obtains the necessary information at runtime are left unspecified. This means that sequence diagram generation can be carried out by the user inside of any context if the user is able to provide the necessary information.

## Architectural Description

The gardener library is designed around a three-tier software architecture. The three subsystems mentioned in the system description correspond broadly to each architecture tier.

The user interface subsystem corresponds to the presentation tier.

The user interface subsystem consists of a single module. The sequence diagram generator module

The parsing subsystem corresponds to the business object tier.

The parsing subsystem consists of three modules:

The reflector module, the parsing module, and the node explorer module.

The input/output subsystem corresponds to the data tier.

The input/output subsystem consists of two modules:

The diagram structure module and the diagram file writer module.

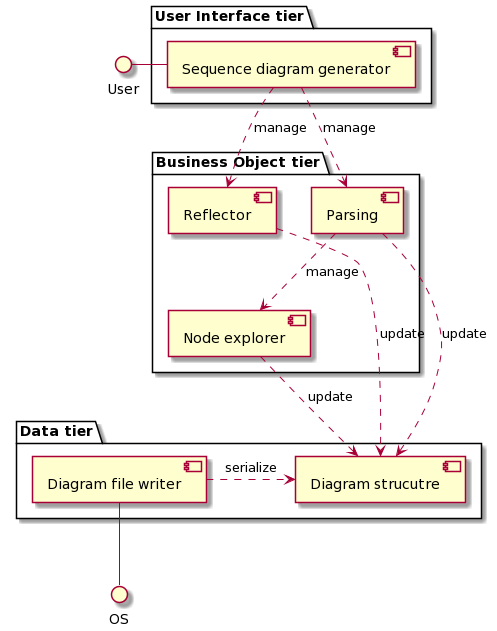


Figure 1: Architechtural description

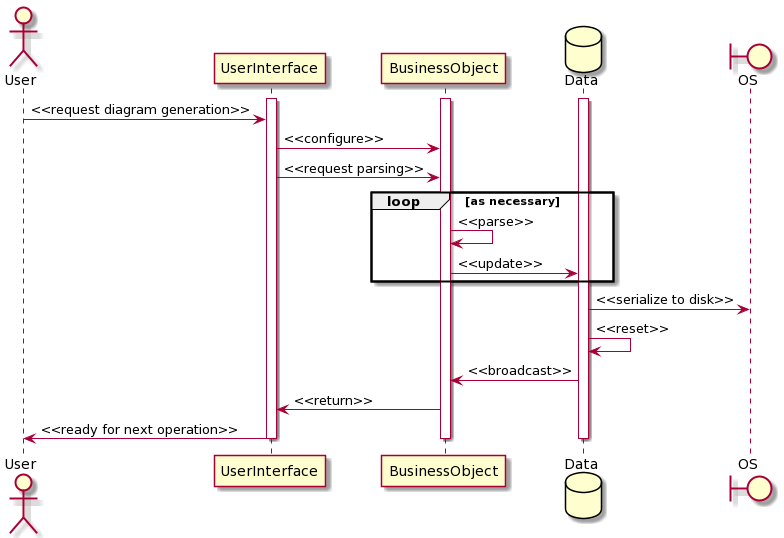
**

Figure 2: Architechtural relations

# USER INTERFACE LAYER DESCRIPTION

The sequence diagram generator module is responsible for letting the end user configure the parsing behavior and provide the necessary information for the creation of the plantUML sequence diagram

The library provides a single implementation of the ISequenceDiagramGenerator interface in the SequenceDiagramGenerator class.

A user wanting to automatically generate a plantUML diagram can instantiate the SequenceDiagramGenerator class inside of his program and call the generateSequenceDiagram method.

The user also has the option of configuring the dependencies used to resolve data types during the parsing of the abstract syntax tree at the business object layer. These dependencies are only used if the appropriate parsing strategy was selected by the user. The specifics of this process are discussed in section 4 of this document

The module is then responsible for managing all the operations carried out by the business logic layer to generate the plantUML sequence diagram image using the information provided by the user.

## Interface

The required external functionality of the UI layer is defined in the ISequenceDiagramGenerator interface.

The external interface of the UI layer consists of the single function generateSequenceDiagram. The role of this function is to allow a user to request the generation of a plantUML sequence diagram.

Prerequisites for this operation include a running instance of an object whose class implements the ISequenceDiagramGenerator interface and the ability of the user to provide the minimum required information for this operation. The specific details of the required information are discussed in section 2.1 of this document.

The expected result of interacting with the UI layer is a text file containing the necessary syntax for the plantUML sequence diagram generation as well as a generated image file of the corresponding sequence diagram.

Generation of the plantUML sequence diagram text file is handled at the UI layer using the external library plantuml

## Structural description

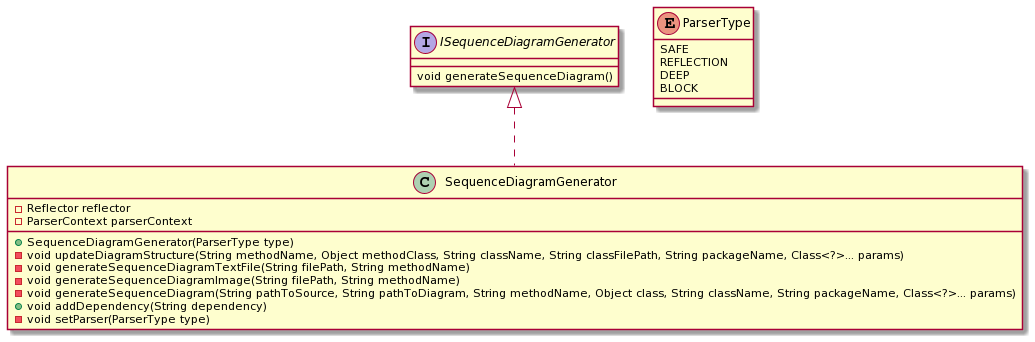


Figure 3: UI layer structure

The goal of the design of the sequence diagram generator is to fully decouple the inner workings of the library from whatever context the user will use the library in while remaining as flexible as possible. To better achieve this goal, the module is constructed around a strategy pattern. This allows the end user to easily swap parsing strategies at runtime while strongly encapsulating the actual business logic implementation behind a layer of abstraction. Specifically, the selection by the user at the UI layer of a parsing strategy as defined inside the ParserType enumeration.

Configuration of the parsing context using the desired parsing strategy is done during object instantiation inside the object constructor. This is intended to ensure that parsing strategies remain consistent during the lifetime of the diagram generator object. However, each sequence diagram generator module instance is completely independent so that more than one generator, each with a different parsing strategy, could be active at any time during runtime.

## Behavioral description

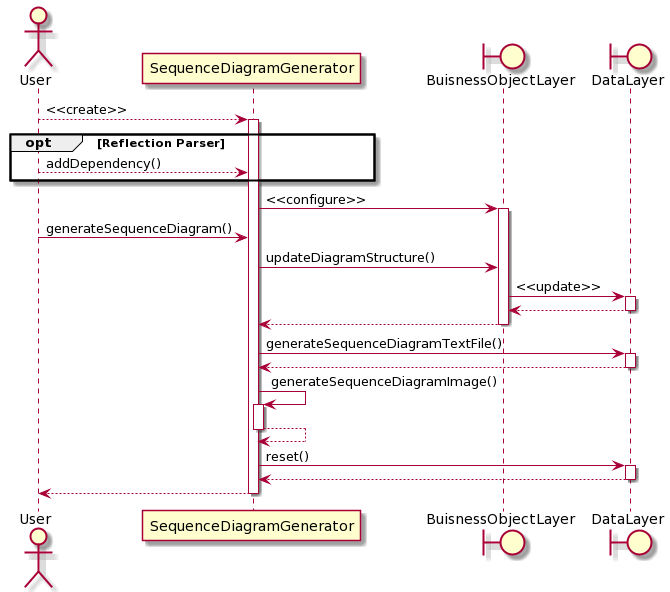
**

Figure 4: UI layer behavior

Error cases:

In case the user fails to provide the necessary information for creation of the plantUML sequence diagram. The program will alert the user and terminate safely. The specific error handling depends on where in the process the error occurred.

Error handling for the reading of the plantUML text file generated by the data layer necessary for the creation of the sequence diagram image file by the plantuml library is carried out at the UI layer.

If an error occurs during the process of reading the plantUML syntax text file, the user will be warned in the form of an exception and the program will terminate safely.

# BUSINESS OBJECT LAYER DESCRIPTION

The reflector module is responsible for using the java reflection API to obtain contextual information on the method/class the user is interested in.

The library provides a single implementation of the IReflector interface in the Reflector class.

This class represents an object that can “reflect” on other java classes using the [Java Reflection](https://www.oracle.com/technical-resources/articles/java/javareflection.html) API. This allows the gardener library to easily obtain some important contextual information on the method the user is interested in making a diagram of. This module represents the first layer of information extraction in the diagram generation process.

The parsing module is responsible for creating the abstract syntax tree with help of the JavaParser library and parsing the resulting tree for information.

The library provides four different implementations of the IJavaParser interface in the classes, SafeJavaParser, ReflectionJavaParser, DeepJavaParser and BlockJavaParser.

The “safe” parser only parses methods that are in the same package as the initial method call the user is interested in. This ensures that contextual information on the methods is always available and that the type of calls can be “safely” resolved. Any calls that aren’t parsed are excluded from the final diagram. Any calls that can’t be resolved are excluded from the final diagram.

The “reflection” parser attempts to resolve the types of methods that are outside of the user package using a variety of different strategies. For example, by extracting information on the type by parsing the method call strings produced by the abstract syntax tree generator, by extracting contextual information from the position of the nodes of the abstract syntax tree or by checking against a list of external packages provided by the user. Any calls that can’t be resolved are show as “lost messages” in the final diagram.

The “block” parser attempts to assign each method call to an execution block by extracting contextual information from the position of the method call nodes of the abstract syntax tree. The execution blocks can then be shown in the final diagram. Any calls that can’t be resolved are excluded from the diagram.

The “deep” parser attempts to resolve method calls “inside” the original method call recursively. Any calls that can’t be resolved are excluded from the diagram.

In general, these classes represent objects that can create an abstract syntax tree from java source code and then parse the resulting tree for information.

The node explorer module is responsible for traversing the nodes of the abstract syntax tree generated by the JavaParser library.

The library provides five implementations of the INodeExplorer interface in the classes, MethodNodeExplorer, ParameterNodeExplorer, VariableNodeExplorer, CatchNodeExplorer and BlockNodeExplorer

In general, these classes represent objects that can extract information from the tree nodes that belong to the abstract syntax tree generated by the JavaParser external software library and then update the data layer accordingly. Different node types are handled with different algorithms by the different classes that extend the NodeExplorer class.

## Interface

The interfaces IJavaParser, IReflector and INodeExplorer represent entry points for the user interface layer to manage the business logic operations. The end user does not need to interact directly with the business object layer and any necessary operations are carried out directly at the user interface layer.

The IReflector interface defines two functions:

The reflectOnClass method extracts information on the context of the class that defines the method the user is interested in.

The reflectOnMethod method extracts information on the context of the method the user is interested in.

Prerequisite for these operations is the proper configuration of the business object layer by the user interface, assuming the user provided the correct information.

The expected result of these operations is the proper updating of the data layer with the information extracted by the reflection API.

The IJavaParser interface defines five functions:

The execute method is a wrapper function for the actual method parsing that provides the option of carrying out some initial configuration steps before the actual parsing of the syntax tree.

The parseFile method is where the necessary information for the generation of the abstract syntax tree is passed on to the external JavaParser software library.

The parseMethod method is where the traversing of the abstract syntax tree provided by the JavaParser library for information extraction is carried out.

The parseMethodNode method is where the individual tree nodes of the abstract syntax tree corresponding to method calls are parsed for information.

The addDependency method is where the dependency list of the parser can be updated.

Prerequisite for these operations is the proper configuration of the business object layer by the user interface, assuming the user provided the correct information.

The expected result of these operations is the proper updating of the data layer with the information extracted from the abstract syntax tree.

The INodeExplorer interface defines a single function, checkNode. This method is responsible for checking the individual tree nodes of the abstract syntax tree provided by the JavaParser external software library.

Prerequisite for this operation is a reference to a node object provided by the JavaParser external library.

The expected result of this operation is the proper updating of the data layer with the information extracted from the individual node of the abstract syntax tree.

## Structural description

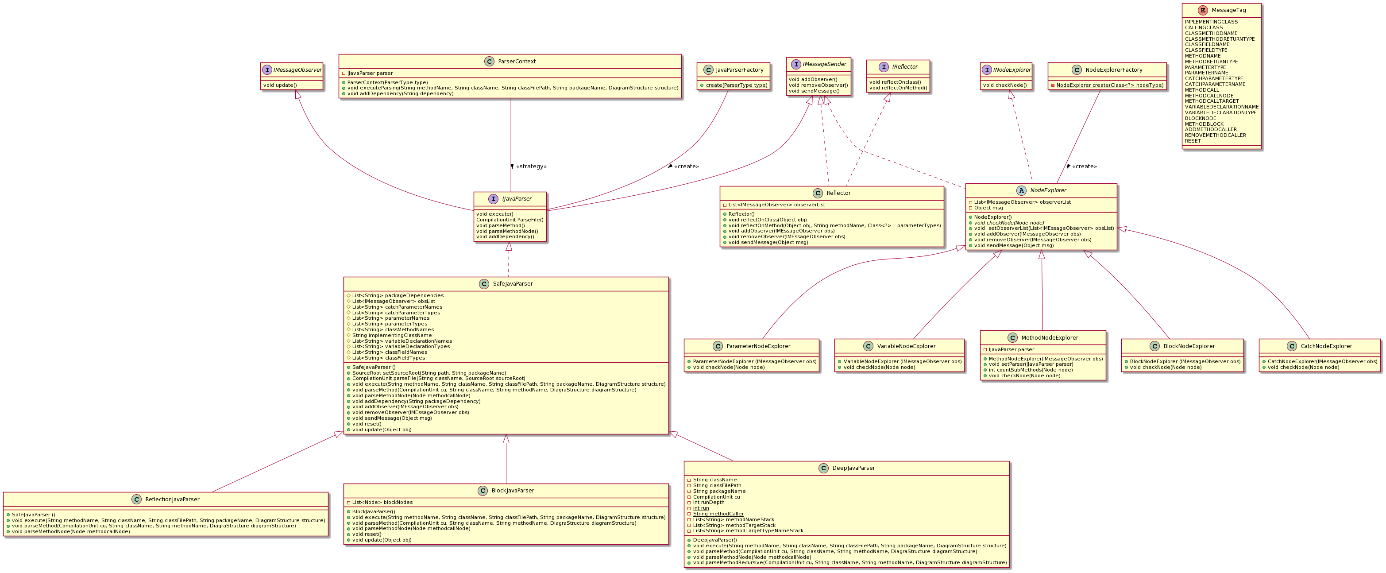
**

Figure 5: BO layer description

The goal of the design of the parser subsystem is to remain as flexible as possible while allowing for the quick implementation of new parsing algorithms that can be employed inside any context in a modular fashion. At the same time, the parsing logic must be strongly decoupled from the outward facing user interface layer and from the specific inner working of the data layer

To better achieve this goal the system is designed around a variety of object-oriented programming patterns.

The abstract syntax tree parser module is constructed around a strategy pattern. This allows the business object layer to be configured for a variety of purposes while presenting an abstracted version of the implemented algorithms to the user interface layer. The configuration of the parsing context is done by the user at runtime by interacting only with the user interface layer.

To simplify the process of implementing new strategies, while allowing for the strong encapsulation of the individual parsing algorithms, the specific parsing objects are created using an object factory.

A second object factory is responsible for the management of the different types of “node explorer” objects used to extract information from the nodes of the abstract syntax tree. This allows new types of node explorer objects to be implemented quickly while encapsulating their properties from the specific parsing strategy used.

To strongly decouple the business object layer from the data layer, communication between the two is done strictly through a messaging system based on subscribers and events. This event system is also used to implement an observer pattern to better handle the one-to-many relation between the database and the many objects that update it. This allows an object to update the database automatically on changes to its inner state while simultaneously propagating the information to any objects that might need it.

## Behavioral description

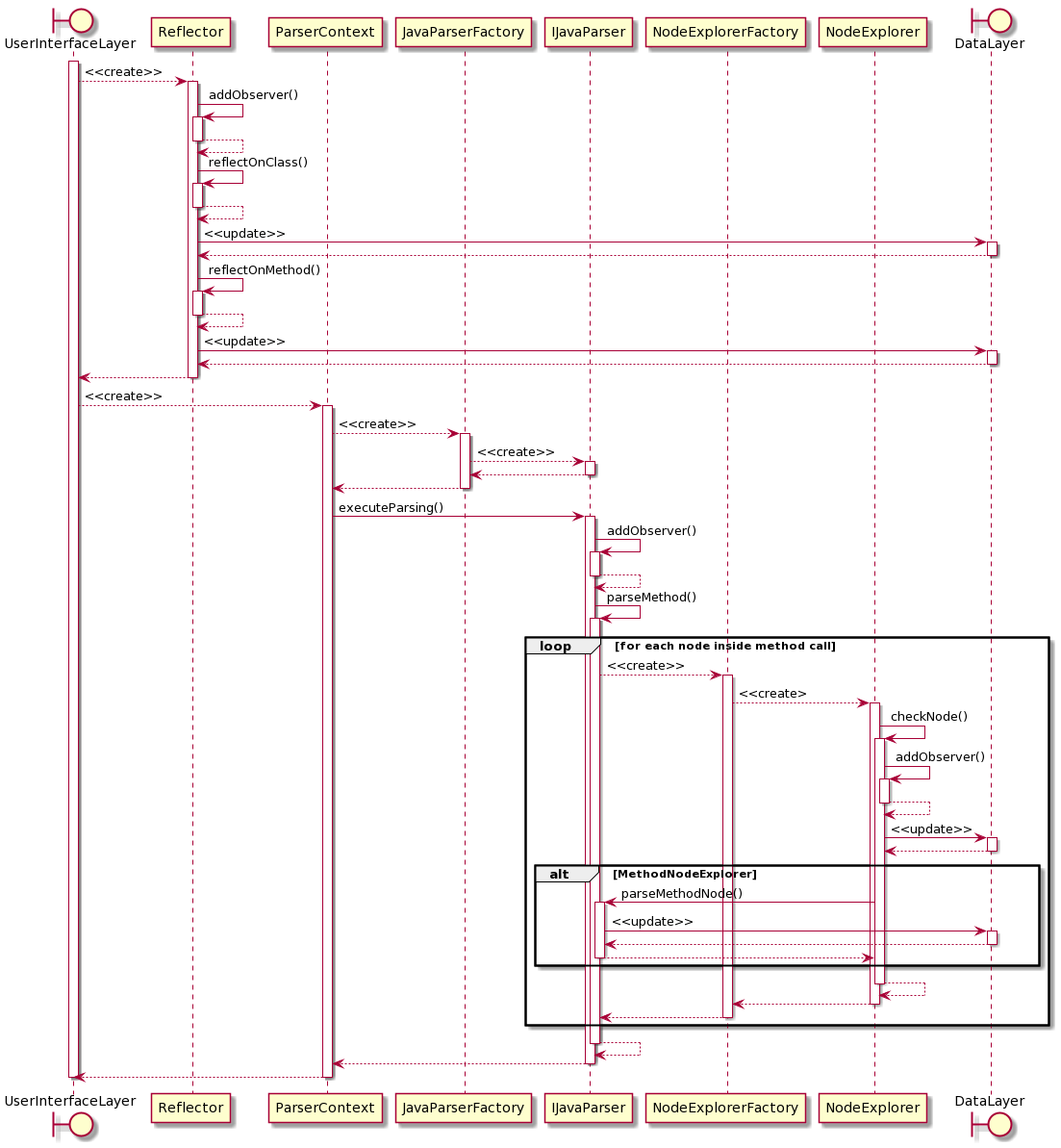


Figure 6: BO layer behavior

Alternative scenarios:

The sequence diagram shows a generified sequence of operations that represents the general flow of operations carried out by the business object layer. The specific sequence of events depends on the IJavaParser and NodeExplorer implementations. See section 4.1 of this document for an overview of the different parsing strategies implemented in the gardener library

Error cases:

In case the user fails to provide the necessary information for creation of the plantUML sequence diagram. The program will alert the user and terminate safely. The specific error handling depends on where in the process the error occurred.

If an error occurs during the process of using the reflection API, the user will be warned in the form of an exception and the program will terminate safely.

If an error occurs during the process of generating the abstract syntax tree, the error will be handled internally by the JavaParser library.

If an error occurs during the process of parsing the syntax tree nodes the user will be warned through a console log and the program will terminate safely.

# DATA LAYER DESCRIPTION

The diagram structure module is responsible for storing all the information extracted at the Business Object layer.

The default implementation in the gardener library is the DiagramStructure class. Most of the information is stored as java strings because the information will eventually be mapped to strings in a text file.

It represents a data structure object that contains all information needed by the plantUML library to generate the desired UML sequence diagram. Any information extracted from a method by the business object layer should be reflected inside this structure.

The diagram file writer module is responsible for writing the information stored in the diagram structure to a text file with the proper plantUML plant syntax necessary to create a plantUML sequence diagram.

The default implementation in the gardener library is the DiagramFileWriter class. The reason this logic heavy operation is part of the data tier and not the business object layer is that it represents a serialization of the database and does not need to relay information back to the user interface layer.

The encapsulation of the file writing operation inside of the data layer also helps decouple the operation from the specific logic used to obtain the information that needs to be serialized.

## Interface

There are no external interfaces exposed to the user for this layer.

Communication between the business object and the data structure layer is carried out through a series of events using the implemented messaging system. This is intended to represent the type of accesses one would use to query information from a traditional external database and to promote the strong decoupling of business object and data layer implementations.

The interface to the external operating system for database serialization is implemented using the standard java IO API.

## Structural description

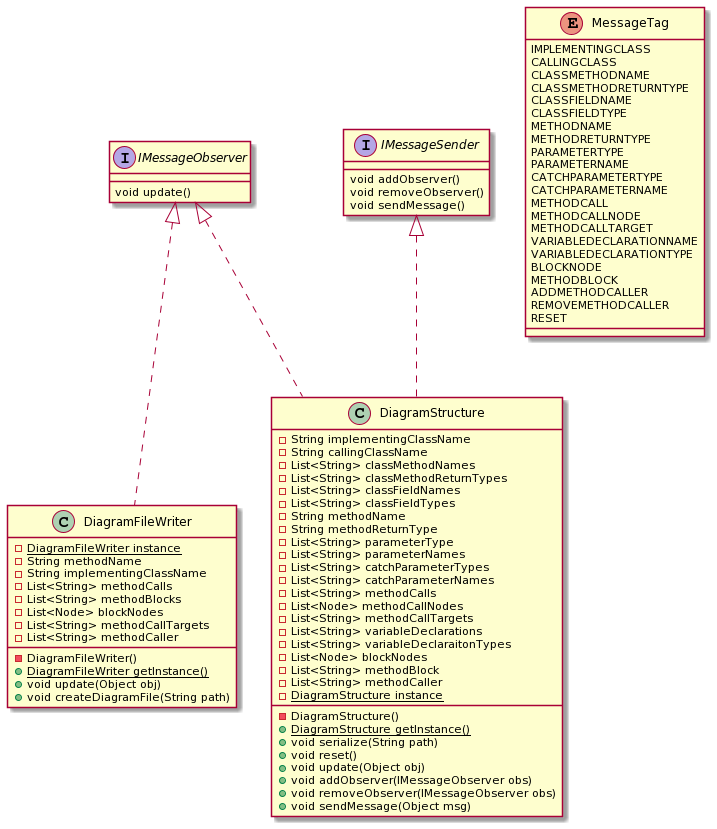


Figure 7: Data layer description

The goal of the design of the data layer is to strongly decouple the data persistence operations from the business object logic used to parse the data from the source code provided by the user. To accomplish this goal communication between the two is done strictly through a messaging system based on subscribers and events.

This event system is also used to implement an observer pattern to better handle the one-to-many relation between the database and the many objects that update it. This allows the database to be updated by a variety of sources while remaining naïve as to their specific implementation. It also allows changes to the database to be automatically propagated to any places in the business logic layer where the information is needed.

To present a consistent data structure to the business object layer the data structure layer is implemented using a singleton pattern. This allows for global access to consistent information across the system without polluting the package namespace with globally accessible variables

Since the diagram file writer is considered an extension of the database structure it is also implemented as a singleton instance.

## Behavioral description

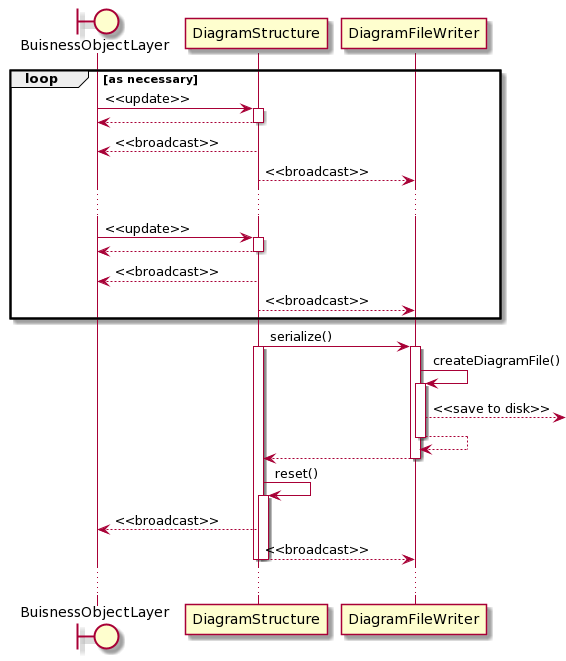


Figure 8: Data layer behavior

Error Cases:

If an error occurs when serializing the database to file. The user will be warned in the form of an exception and the program will terminate safely.

# External references

General Information about the external library JavaParser used for generating abstract syntax trees can be obtained at [JavaParser - Home](https://javaparser.org/).

The specific functionality of the JavaParser library is documented at [javaparser-core 3.24.0 javadoc (com.github.javaparser)](https://javadoc.io/doc/com.github.javaparser/javaparser-core/latest/index.html)

General Information about plantUML and the plantuml external library used for generating the sequence diagram images can be obtained at [plantuml.com](https://plantuml.com/)

The software requirement specifications document should always be provided as a package together with this document.

# Annex

The following documents and files should always be provided as a package together with this document:

Architectural description UML diagrams:

Arch.png

Architectural sequence UML diagrams:

ArchSeq.png

BO layer description UML diagrams:

BOlayer.png

BO layer sequence UML diagrams:

BOlayerSEQ.png, BOlayerSEQ.txt

Data layer description UML diagrams:

DOlayer.png

DO layer sequence UML diagrams:

DOlayerSEQ.png

UI layer description UML diagrams:

UIlayer.png

UI layer sequence UML diagrams:

UIlayerSEQ.png

Software Requirements Specification document:

SRS.docx