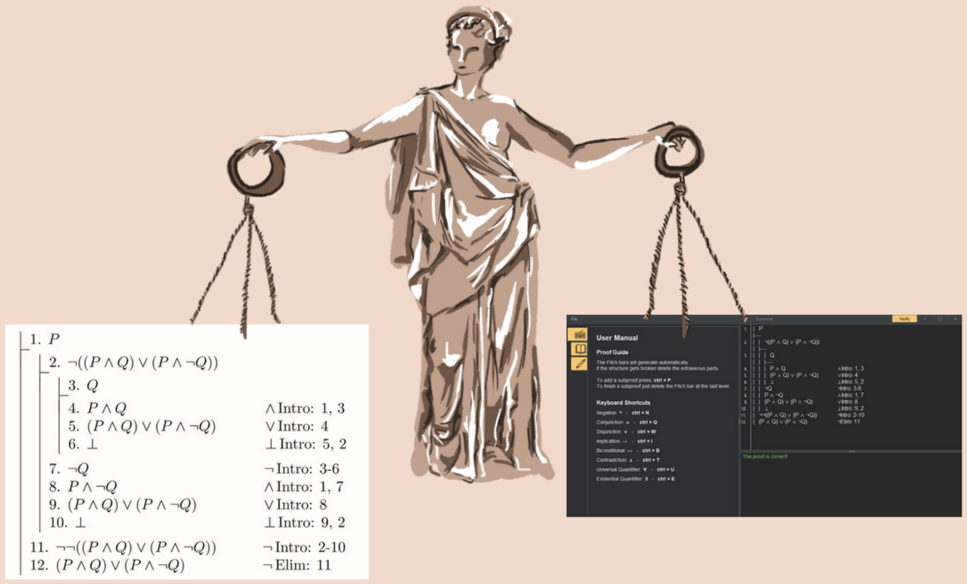
Modified date: 22-08-2023
Total Page Number: 12
Version: 1.0

Original Author: Andrei Girjoaba

Eunomia - Design Document

Everything You Need to Know before Coding



1. P
2. $\neg((P \wedge Q) \vee (P \wedge \neg Q))$
3. Q
4. $P \wedge Q$ — $\wedge$Intro: 1, 3
5. $(P \wedge Q) \vee (P \wedge \neg Q)$ — $\vee$Intro: 4
6. $\bot$ — $\bot$Intro: 5, 2
7. $\neg Q$ — $\neg$Intro: 3-6
8. $P \wedge \neg Q$ — $\wedge$Intro: 1, 7
9. $(P \wedge Q) \vee (P \wedge \neg Q)$ — $\vee$Intro: 8
10. $\bot$ — $\bot$Intro: 9, 2
11. $\neg\neg((P \wedge Q) \vee (P \wedge \neg Q))$ — $\neg$Intro: 2-10
12. $(P \wedge Q) \vee (P \wedge \neg Q)$ — $\neg$Elim: 11

User Manual

Proof Guide

The Fitch bars will generate automatically.
If the structure gets broken delete the extraneous parts.

To add a subproof press: **ctrl + P**.
To finish a subproof just delete the Fitch bar at the last level.

Keyboard Shortcuts

Negation: ¬ - **ctrl + N**
Conjunction: ∧ - **ctrl + Q**
Disjunction: ∨ - **ctrl + W**
Implication: → - **ctrl + I**
Biconditional: ↔ - **ctrl + B**
Contradiction: ⊥ - **ctrl + T**
Universal Quantifier: ∀ - **ctrl + U**
Existential Quantifier: ∃ - **ctrl + E**

```
1.  | P
    |---
2.  | | ¬((P ∧ Q) ∨ (P ∧ ¬Q))
    | |---
3.  | | | Q
    | | |---
4.  | | | P ∧ Q                  ∧Intro: 1, 3
5.  | | | (P ∧ Q) ∨ (P ∧ ¬Q)     ∨Intro: 4
6.  | | | ⊥                      ⊥Intro: 5, 2
7.  | | ¬Q                       ¬Intro: 3-6
8.  | | P ∧ ¬Q                   ∧Intro: 1, 7
9.  | | (P ∧ Q) ∨ (P ∧ ¬Q)       ∨Intro: 8
10. | | ⊥                        ⊥Intro: 9, 2
11. | ¬¬((P ∧ Q) ∨ (P ∧ ¬Q))     ¬Intro: 2-10
12. | (P ∧ Q) ∨ (P ∧ ¬Q)         ¬Elim: 11
```

The proof is correct!

Contribution Record

Contributor	Update Date	Version	Description
Andrei Girjoaba	30-07-2023	0.0	Creation
Andrei Girjoaba	22-08-2023	1.0	First Complete Version

Contents

1 What is Eunomia?

Eunomia is an educational software that is aiding students in writing first order logic proofs in Fitch Style. It basically is a compiler for first order logic.

Fitch Proofs

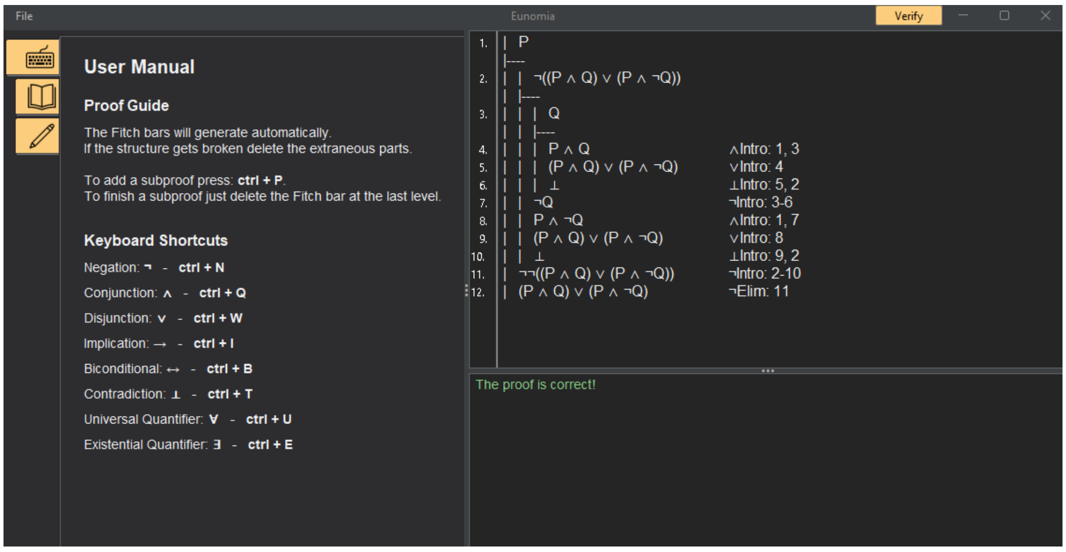
The style in which the proofs are aimed to be written are described in the course book: ***Language, Proof and Logic*** *by Dave Barker-Plummer, Jon Barwise and John Etchemendy*

UI Inspiration

It draws inspiration from Code Editors like Visual Studio Coded. That was a deliberate choice since *Introduction to Logic* is usually a first year subject in Computing Science, Artificial Intelligence, Physics, etc. Bachelors, it is a good idea to familiarize students with such environments.

Open Source

Since it is an educational tool, the more education it provides the better. Everyone is more than welcome to use (or change it to their own desires) if they feel it is useful. I would say this is a way to evaluate its success and being open source helps with obtaining this goal.



User Manual

Proof Guide

The Fitch bars will generate automatically.
If the structure gets broken delete the extraneous parts.

To add a subproof press: **ctrl + P**.
To finish a subproof just delete the Fitch bar at the last level.

Keyboard Shortcuts

Negation: ¬ - **ctrl + N**
Conjunction: ∧ - **ctrl + Q**
Disjunction: ∨ - **ctrl + W**
Implication: → - **ctrl + I**
Biconditional: ↔ - **ctrl + B**
Contradiction: ⊥ - **ctrl + T**
Universal Quantifier: ∀ - **ctrl + U**
Existential Quantifier: ∃ - **ctrl + E**

```
1.  |  P
    |----
2.  |  |  ¬((P ∧ Q) ∨ (P ∧ ¬Q))
    |  |----
3.  |  |  |  Q
    |  |  |----
4.  |  |  |  P ∧ Q                      ∧Intro: 1, 3
5.  |  |  |  (P ∧ Q) ∨ (P ∧ ¬Q)         ∨Intro: 4
6.  |  |  |  ⊥                          ⊥Intro: 5, 2
7.  |  |  ¬Q                            ¬Intro: 3-6
8.  |  |  P ∧ ¬Q                        ∧Intro: 1, 7
9.  |  |  (P ∧ Q) ∨ (P ∧ ¬Q)            ∨Intro: 8
10. |  |  ⊥                             ⊥Intro: 9, 2
11. |  ¬¬((P ∧ Q) ∨ (P ∧ ¬Q))           ¬Intro: 2-10
12. |  (P ∧ Q) ∨ (P ∧ ¬Q)               ¬Elim: 11
```

The proof is correct!

Figure 1: Screenshot of Eunomia software as version: 0.1.

2 Compiler

The main logic part of the application is a compiler that is able to identify all the errors occurring in a FOL proof. The compiler was created using ANTLR (ANother Tool for Language Recognition).

2.1 Grammar

The grammar defines the structure of a proof in its entirety. The main components are:

- **Proof lines:** they contain the line number, logical sentence itself and its justification.
- **Subproofs:** they contain either other subproofs or proof lines.

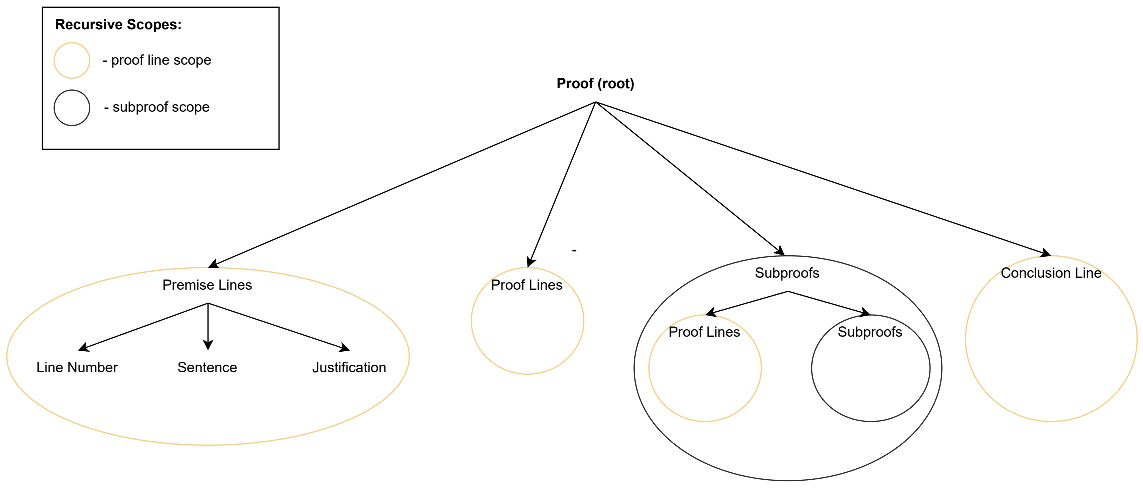


Figure 2: A schema describing the components of the grammar.

For a complete definition of the grammar see the *ProofGrammar.g4* file inside the modelLogic module.

2.2 Syntax Analyzer - ANTLR

The syntax analysis is done by the parser generated with ANTLR, which is a powerful tool which is able to generate a parser for the grammar we defined.

2.3 Semantic Analyzer - ANTLR

ANTLR also generates a *BaseVisitor* which is built upon the Visitor Design Pattern. We extend this class in order to semantically analyze our proof.

The semantics are analyzed as follows:

1. We encounter a proof line and store its the relevant information. (To asses the correctness of a proof line, we only need information from the previous proof lines.)
2. When we encounter a justification, we look into the stored information of the previous proof lines and establish if the current line follows.
3. If we encounter a semantic mistake in the proof we store it with the line number associated to it.
4. We repeat steps 1-3 until the whole proof is traversed.
5. If there are no errors stored, the proof is correct, otherwise it is wrong.

2.4 Input/Output

As input, the compiler can take either a String or the path to a *.txt* file.

As output, the compiler uses a *Proof Manager*. The Proof Manager keeps track of the correctness of each proof line. It also keeps track of the encountered syntax errors during parsing. If there are any syntax errors the semantics analysis will not proceed.

2.5 Class Diagram: Compiler

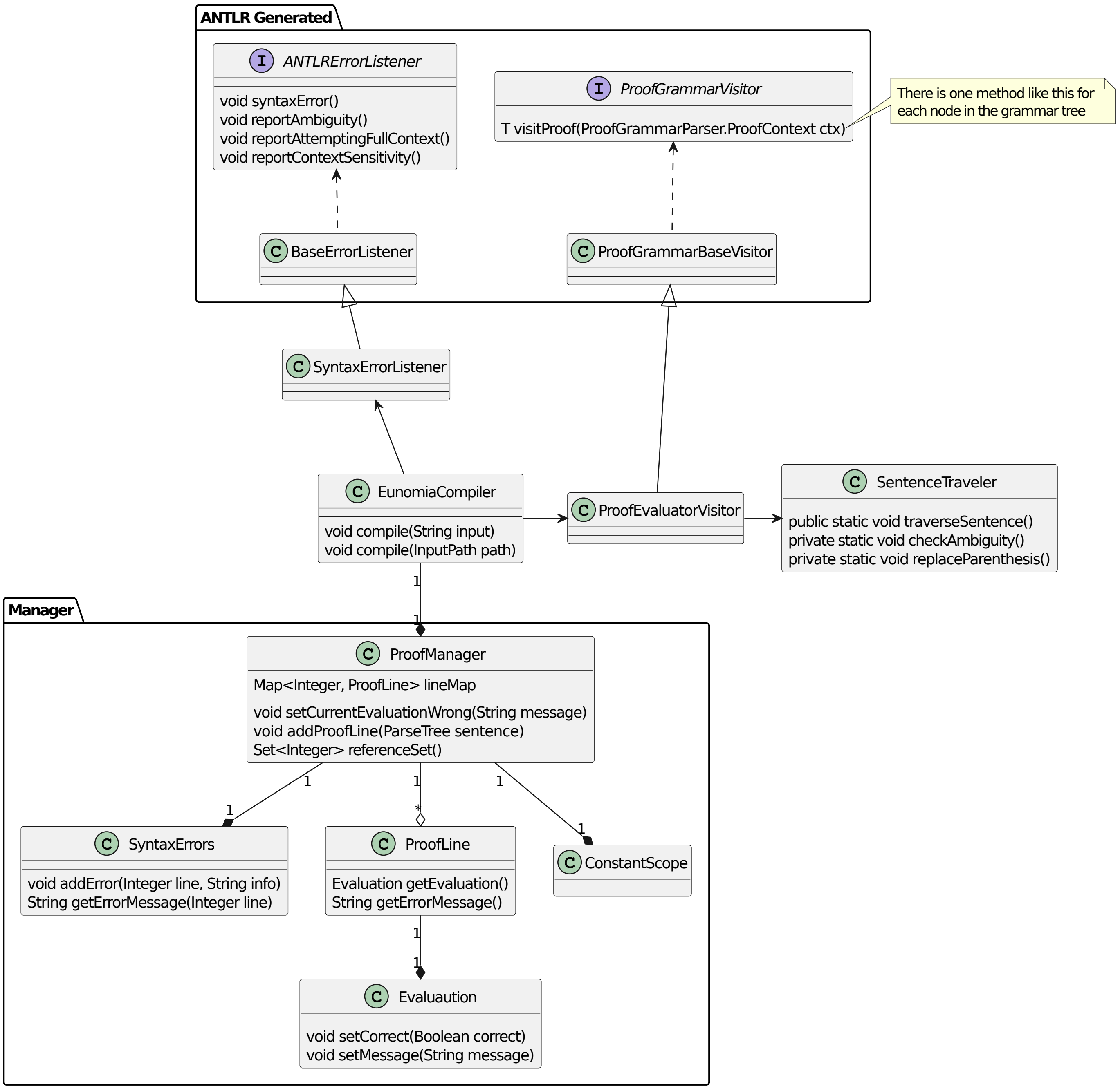

ANTLR Generated
I ANTLRErrorListener
void syntaxError()
void reportAmbiguity()
void reportAttemptingFullContext()
void reportContextSensitivity()
I ProofGrammarVisitor
T visitProof(ProofGrammarParser.ProofContext ctx)
There is one method like this for each node in the grammar tree
C BaseErrorListener
C ProofGrammarBaseVisitor
C SyntaxErrorListener
C EunomiaCompiler
void compile(String input)
void compile(InputPath path)
C ProofEvaluatorVisitor
C SentenceTraveler
public static void traverseSentence()
private static void checkAmbiguity()
private static void replaceParenthesis()
Manager
C ProofManager
Map<Integer, ProofLine> lineMap
void setCurrentEvaluationWrong(String message)
void addProofLine(ParseTree sentence)
Set<Integer> referenceSet()
C SyntaxErrors
void addError(Integer line, String info)
String getErrorMessage(Integer line)
C ProofLine
Evaluation getEvaluation()
String getErrorMessage()
C ConstantScope
C Evaluaution
void setCorrect(Boolean correct)
void setMessage(String message)
1
*

Figure 3: Class Diagram of the Compiler. EunomiaCompiler is the top-most class.

3 Architecture - Model-View-Controller

We also require a user interface. The compiler is just one part of the application after all! It is a good idea to decouple the two as they are two very distinct components. This allows for easy parallel development on either side.

To achieve this I chose the Model-View-Controller (MVC) architecture. The java project is organized in three modules:

- **modelLogic:** The compiler.
- **fitchTextEditor:** The UI.
- **controller:** The entry point of the application and what establishes the communication between the UI and the compiler.

3.1 Dependencies

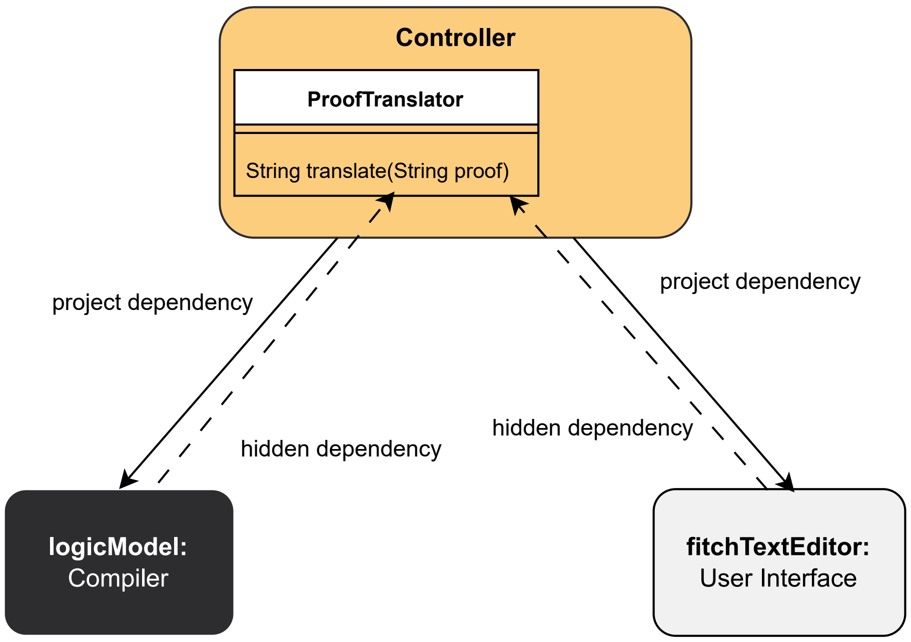


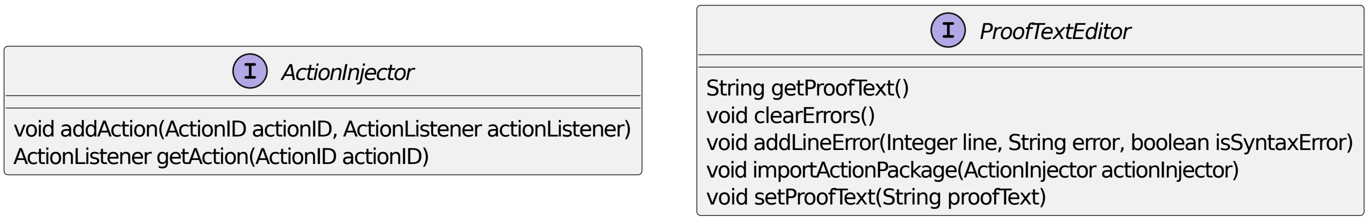
Figure 4: Project architectural dependencies.

The controller uses directly the UI and the Compiler. These dependencies are directly dictated by the imports in the Controller module.

! Important: There is one hidden dedpendency inside the Controller. That is in the *ProofTranslator* class. This class has the responsibility of translating whatever the UI provides to whatever the Compiler expects. This means that either when the grammar of the compiler is changed or the way the proof is written in the UI is changed, this class **must** also be changed!

3.2 Controller-View Communication

The controller also provides logic for how some actions inside the view should execute. The controller also expects a specific output from the view. This expected functionality is provided by two interfaces that the view must implement.



```
void addAction(ActionID actionID, ActionListener actionListener)
ActionListener getAction(ActionID actionID)
```

```
String getProofText()
void clearErrors()
void addLineError(Integer line, String error, boolean isSyntaxError)
void importActionPackage(ActionInjector actionInjector)
void setProofText(String proofText)
```

Figure 5: The interfaces that the view must implement.

The *ActionInjector* is used to inject actions into the view.

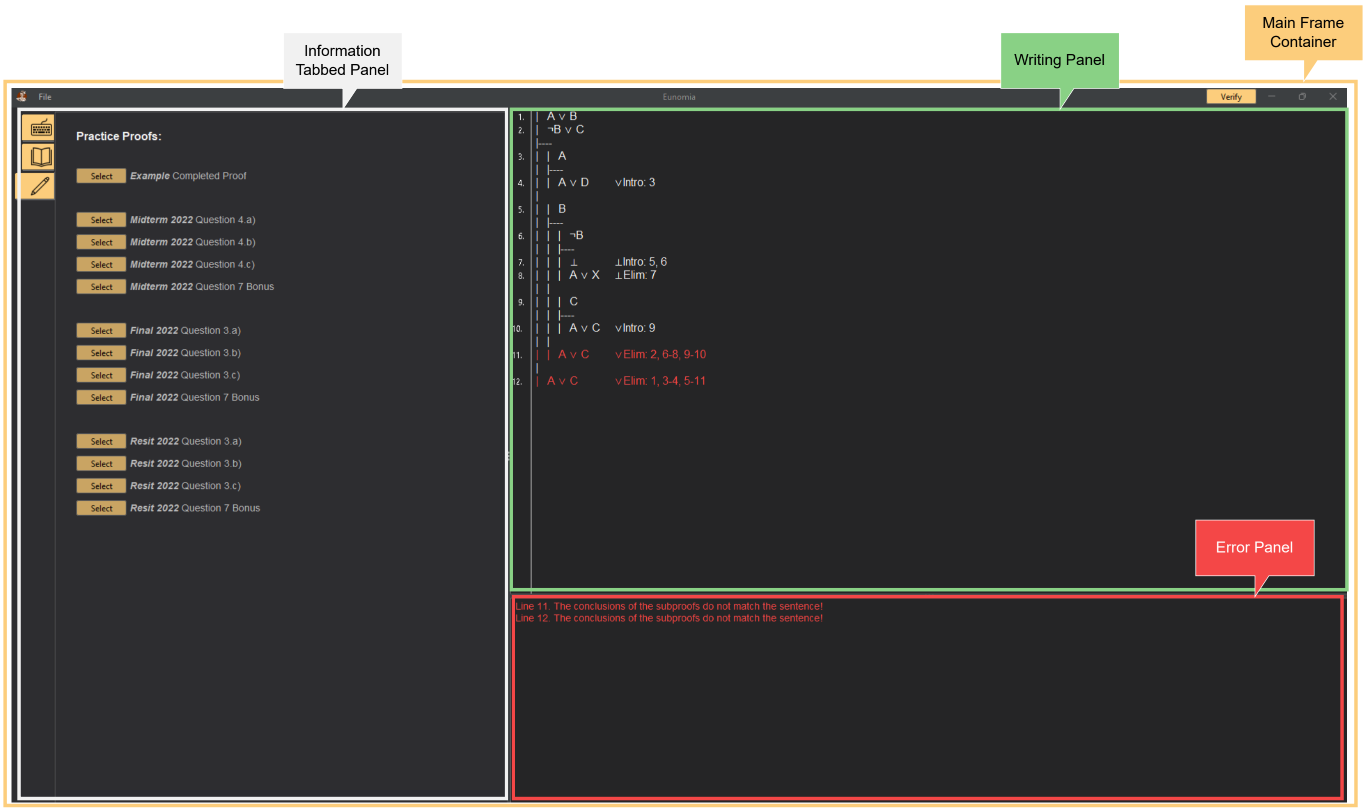
The *ProofTextEditor* is used to get the proof from the editor, set it and manage errors.

4 User Interface

The User Interface (UI) uses a similar layout to most code editors. The frame contains three main panels: the writing panel (where the proof is being written), the error panel (where the errors are displayed) and the information panel (where different utilities are information are presented to the user).

4.1 Techonology: Java Swing

The current view of the UI is built using Java Swing. Java Swing is based on the composite pattern, therefore the frame and its inner panels are containers for all the other components or inside panels.



Practice Proofs:

Example Completed Proof

Midterm 2022 Question 4.a)
Midterm 2022 Question 4.b)
Midterm 2022 Question 4.c)
Midterm 2022 Question 7 Bonus

Final 2022 Question 3.a)
Final 2022 Question 3.b)
Final 2022 Question 3.c)
Final 2022 Question 7 Bonus

Resit 2022 Question 3.a)
Resit 2022 Question 3.b)
Resit 2022 Question 3.c)
Resit 2022 Question 7 Bonus

```
1.  |  A ∨ B
2.  |  ¬B ∨ C
    |----
3.  |  |  A
    |  |----
4.  |  |  A ∨ D       ∨Intro: 3
    |
5.  |  |  B
    |  |----
6.  |  |  |  ¬B
    |  |  |----
7.  |  |  |  ⊥        ⊥Intro: 5, 6
8.  |  |  |  A ∨ X    ⊥Elim: 7
    |  |
9.  |  |  |  C
    |  |  |----
10. |  |  |  A ∨ C    ∨Intro: 9
    |  |
11. |  |  A ∨ C       ∨Elim: 2, 6-8, 9-10
    |
12. |  A ∨ C          ∨Elim: 1, 3-4, 5-11
```

```
Line 11. The conclusions of the subproofs do not match the sentence!
Line 12. The conclusions of the subproofs do not match the sentence!
```

Figure 6: Main containers of the UI.

4.2 Class Diagram: UI

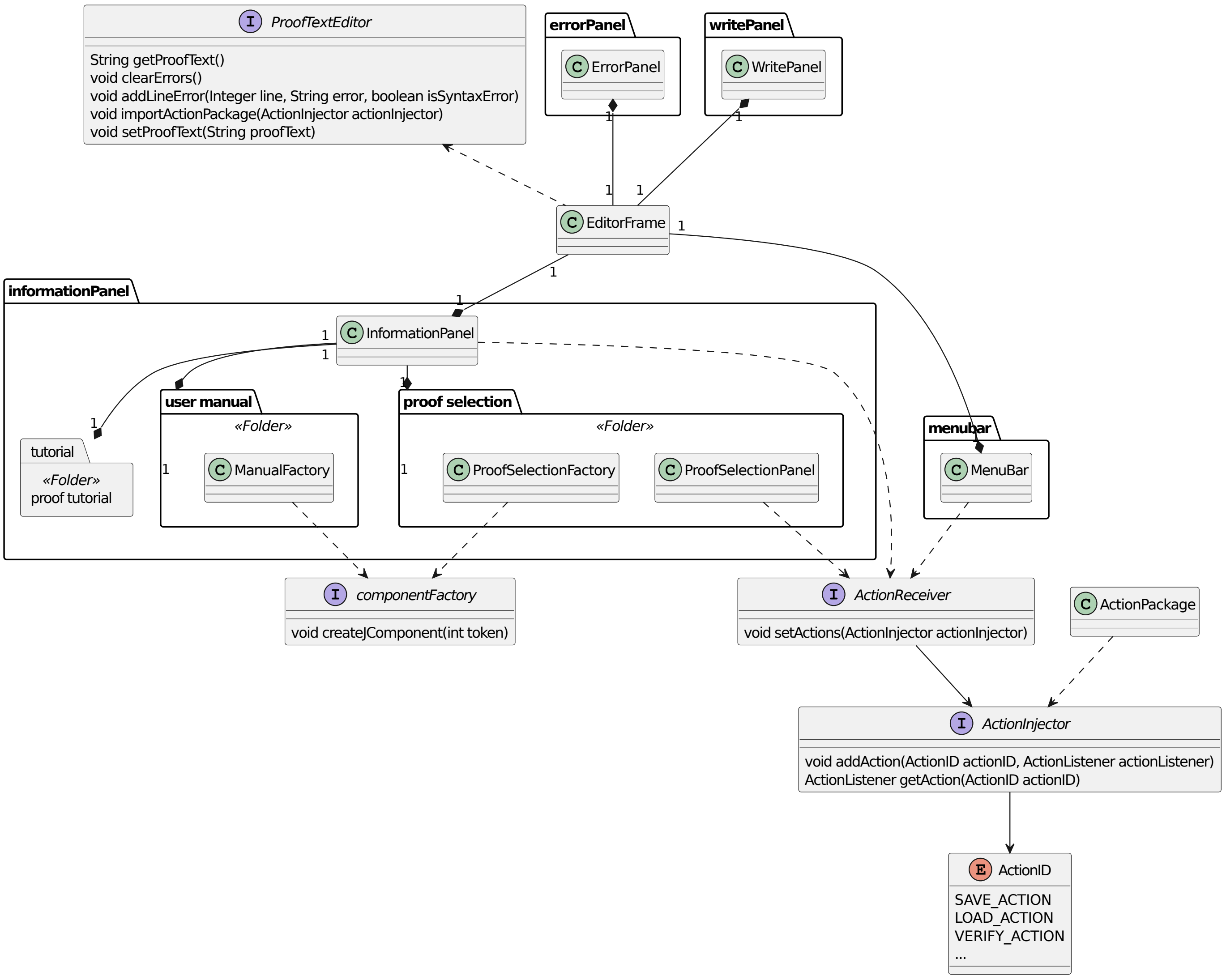


Figure 7: Class Diagram of the UI. EditorFrame is the main container.

4.3 Factories

Factories are used for quick customization of panels. A factory will provide a JComponent for each descriptive token which is made public in the specific factory. This allows for quick customization and reordering of the compoents in a panel.

Create a factory whenever a panel contains multiple components that can be rearranged.

5 Testing

Thorough testing was conducted for the compiler. On the UI side only user testing was implemented so far.

5.1 Compiler Testing

5.1.1 Logical Laws

The testing of the compiler follows a strict structure. The tested logical laws are:

- Contradiction
- Identity
- Negation
- Conjunction
- Disjunction
- Implication
- Biconditional
- Universal Quantifier
- Existential Quantifier

Each law has a corresponding class and in every class its introduction and elimination rule is tested.

Each rule is tested for all positives and negatives I could think of. A high coverage of everything that could go wrong with a rule was attempted. False negatives and false positives must be kept in mind.

The testing is conducted by loading a proof from the *resources* folder and verifying that each line that is supposed to be wrong, is actual wrong.

Note: This is great test driven development on the compiler side.

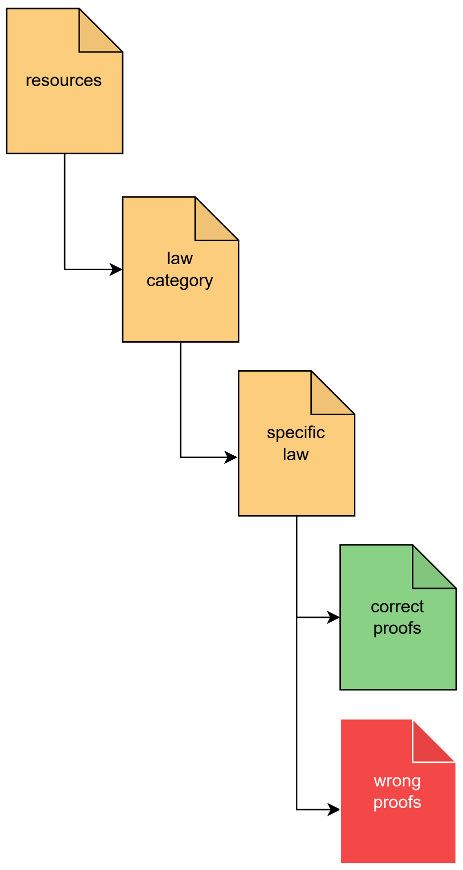


Figure 8: Folder structure of the testing resources.

5.1.2 Syntax and Structure

There is more to a compiler the compiler than just the logical rules. The proof must follow a specific structure with its subproofs, premises, ambiguous order of operations, etc.

A similar approach was executed when testing these aspects. A proof was loaded from the *resources* and it was verified that it fails only when is supposed to fail.

5.2 UI Testing

Before each release, user testing is done on multiple individuals.

Appendix A: FAQ

How it came to be?

It started as a project at the University of Groningen. Upon completion I though it would be nice to make it open source and see if anybody will actually use it.

Why this name?

At the University of Groningen another software in use is called Themis. Eunomia is the daughter of Themis in greek mythology. Plus Eunomia is the goddess of law, governance, and good order. Not quite logic laws but it was good enough for a name.

Appendix B: Development

Currently, this is an open source project. You can find the repository at https://github.com/Girjoaba/Eunomia/tree/main.