TOOL FOR RAPID SOFTWARE ARCHITECTURE DESIGN WITH SPLIT-N-JOIN ACTIONS

Antonios Peris

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

TOOL FOR RAPID SOFTWARE ARCHITECTURE DESIGN WITH SPLIT-N-JOIN ACTIONS

Thesis submitted by
Antonios Peris
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Antonios Peris

Committee approvals: _____

Anthony Savidis
Professor, Thesis Supervisor

_____

Konstantinos Magoutis
Associate Professor, Committee Member

_____

Dimitrios Gramenos
Principal Researcher, Committee Member

Department approval: _____

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, September 2022

TOOL FOR RAPID SOFTWARE ARCHITECTURE DESIGN WITH SPLIT-N-JOIN ACTIONS

Abstract

Software architecture designing is a stage of vital importance in software development, as its product acts as a bridge between the requirements and implementation. A well-constructed architecture offers the software, both short and long-term benefits. It contains the system's functional roles and its operational features that are represented graphically by components and operations respectively. Components often need to communicate and therefore a link between them is created. The architecture is being altered repeatedly until it reaches its ultimate form, in which every operation is attached to a component and no further action can be done upon components.

This procedure is defined by its simplicity and abstraction. However, most of the current software architecture designing tools do not comply with these characteristics. Instead, they emphasize on graphical and informational details that might be currently unknown to the user or may be volatile and will be consolidated in the latter stages of development. As a consequence, not only do these details become sometimes overwhelming and tiresome but also the software architect is misled and stalled unintentionally from achieving their real goal.

To offer a solution to this problem, we introduce a rapid software architecture designing tool, which reduces to the bare minimum the informational and graphical technicalities. More specifically, it requires only a name and an optional description for the creation of any component, link, or operation in contrast with already available designing tools. Each type of these elements has only one graphical representation. Moreover, our tool provides specific software architectural abilities, in which the user manipulates the elements until all the system's requirements are fulfilled and the architecture obtains its final form. In this manner,

the user saves a considerable amount of time and simultaneously focuses on making architectural decisions.

ΕΡΓΑΛΕΙΟ ΓΡΗΓΟΡΟΥ ΠΡΟΣΔΙΟΡΙΣΜΟΥ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΜΕ ΕΝΕΡΓΕΙΕΣ SPLIT ΚΑΙ JOIN

Περίληψη

Η σχεδίαση αρχιτεκτονικής λογισμικού είναι ένα στάδιο ζωτικής σημασίας για την ανάπτυξη ενός συστήματος, καθώς το προϊόν της λειτουργεί ως γέφυρα μεταξύ των απαιτήσεων και της υλοποίησής του. Μια σωστά δομημένη αρχιτεκτονική προσφέρει στο λογισμικό τόσο βραχυπρόθεσμα όσο και μακροπρόθεσμα οφέλη. Αρχικά, περιέχει τους λειτουργικούς ρόλους και τα χαρακτηριστικά του σχεδιαζόμενου συστήματος, υπό την μορφή δομικών στοιχείων και συναρτήσεων αντίστοιχα. Συχνά, τα δομικά στοιχεία χρειάζεται να επικοινωνούν μεταξύ τους. Σε αυτήν την περίπτωση, δημιουργείται ένα ευθύγραμμο τμήμα που ενώνει αυτά τα στοιχεία μεταξύ τους. Η δομή της αρχιτεκτονικής μεταβάλλεται επαναλαμβανόμενα, μέχρις ότου να αποκτήσει την τελική της μορφή, κατά την οποία καμία περαιτέρω πράξη δεν χρειάζεται να πραγματοποιηθεί πάνω στα δομικά της στοιχεία και κάθε συνάρτηση έχει τοποθετηθεί σε ένα δομικό στοιχείο.

Αυτή η διαδικασία χαρακτηρίζεται από την απλότητά της και την χρήση αφηρημένων εννοιών. Ωστόσο, τα περισσότερα σύγχρονα εργαλεία για σχεδίαση αρχιτεκτονικής λογισμικού δεν συμβαδίζουν με αυτά τα χαρακτηριστικά. Εν αντιθέσει, δίνουν έμφαση σε λεπτομέρειες όπου αφορούν τις πληροφορίες και την μορφή των στοιχείων, όπου σε αυτήν την φάση της ανάπτυξης του συστήματος μπορεί να είναι άγνωστες ή ευμετάβλητες και θα μονιμοποιηθούν στα επόμενα στάδια της υλοποίησης. Κατά συνέπεια, οι λεπτομέρειες αυτές είναι φορτικές για τον αρχιτέκτονα λογισμικού, ο οποίος ακούσια αποπροσανατολίζεται και καθυστερεί την επίτευξη του πραγματικού του στόχου.

Ως λύση σε αυτό το ζήτημα, δημιουργήσαμε ένα γρήγορο εργαλείο σχεδίασης αρχιτεκτονικής λογισμικού, στο οποίο οι τεχνικές λεπτομέρειες που αφορούν τα γραφικά στοιχεία και τα δεδομένα τους έχουν μειωθεί στο ελάχιστο. Πιο συγκεκριμένα, για την

δημιουργία οποιουδήποτε στοιχείου, χρειάζεται μόνο η συμπλήρωση ενός ονόματος και προαιρετικά μιας περιγραφής, σε αντίθεση με την προαναφερόμενη κατηγορία εργαλείων. Ο κάθε τύπος στοιχείου έχει μόνο μία γραφική αναπαράσταση. Επιπρόσθετα, το εργαλείο μας περιλαμβάνει συγκεκριμένες δυνατότητες, στις οποίες ο χρήστης τροποποιεί τα δομικά στοιχεία και τις συναρτήσεις, μέχρις ότου όλες οι απαιτήσεις του συστήματος να πληρούνται και η αρχιτεκτονική να έχει λάβει την τελική της μορφή. Κατά αυτόν τον τρόπο, εξοικονομεί χρόνο και ταυτόχρονα επικεντρώνεται στην λήψη σημαντικών, για την τρέχουσα φάση, αποφάσεων.

Acknowledgments

Firstly, I would like to thank my thesis supervisor, Professor Anthony Savidis of the Computer Science Department at the University of Crete, for trusting me with this interesting thesis subject and for his continual assistance. I am also grateful to Principal Researcher Dimitrios Gramenos and Associate Professor Konstantinos Magoutis for participating in the supervisory committee. Also, I would like to thank the Computer Science Department for offering a high level academic education and for equipping me with the right skills to carry through this thesis. Furthermore, I want to thank each member of the PLATO individually for their valuable advice and for our productive conversations. Finally, most of all, I would like to thank my family for all their love and for being by my side every step of the way. Without them, I would not be who I am today.

Στην οικογένειά μου

Contents

Contents.....13

List of Figures.....17

Introduction.....19

1.1. Background.....19

1.1.1. Components, Operations & Links.....19

1.1.2. Software Architecture Designing Process.....20

1.1.3. Rapid Software Architecture Designing Process.....21

1.2. Problem Definition & Objectives.....22

1.3. Thesis Structure.....23

2. Related Work.....25

2.1. Detailed Specification Tools.....25

2.2. Rapid Designing Tools.....27

3. System Overview and Architecture.....30

3.1. System's Architecture.....30

3.2. Instance Generator.....31

3.3. Blackboard.....31

3.3.1. Item Holder.....32

3.3.2. Layer Holder.....32

3.3.3. Action Holder.....32

3.4. Observer.....33

4. Software Architecture Designing Actions.....36

4.1. Component Actions.....36

4.1.1 New & Delete.....36

4.1.2 Link & Unlink.....38

4.1.3 Insert & Delete Component's Sub-architecture.....39

4.1.4 Join Components.....40

4.1.5 Extend & Collapse.....40

4.1.6 Copy & Paste.....41

4.2. Operation Actions.....42

4.2.1 Creation & Deletion.....42

4.2.2 Set & Reset.....43

4.2.3 Move.....45

4.2.4 Split.....45

4.3. Layer Actions.....46

4.3.1 Creation & Deletion.....46

4.3.2 Layer Change.....47

4.3.3 Move Components Across Layers.....47

4.3.4 Layer Info.....48

4.4. Project Actions.....49

4.4.1 Save & Load Project.....49

4.4.2 New Project.....50

5. Software Architecture Visualization & Configuration.....53

5.1. User Interface.....53

5.2. Implementation.....54

5.2.1 Workspace.....54

5.2.2 Toolbar.....55

5.2.3 Operations' Table.....56

5.2.4 Hierarchy Tree.....57

5.3. Context Menus.....57

5.4. Configurations.....59

5.4.1 Component Settings.....59

5.4.2 Operation Settings.....61

5.4.3 Link Settings.....61

6. Case Study.....64

6.1. Super Mario Game Engine's Architecture.....64

6.1.1. Components.....64

6.1.2. Operations.....65

6.1.3. Final Result.....66

6.2. Compiler's Architecture.....67

6.2.1. Components.....67

6.2.2. Operations.....68

6.2.3. Final Result.....69

6.3. Rapid Software Architecture Designer's Architecture.....69

6.3.1. Components.....70

6.3.2. Operations.....71

6.3.3. Final Result.....76

7. Conclusion and Future Work.....79

Bibliography.....82

List of Figures

Figure 1: System's Architecture.....30
Figure 2: Actions' JSON.....33
Figure 3: Undo and Redo methods.....33
Figure 4: Component Creation.....37
Figure 5: Component Deletion.....37
Figure 6: Link Creation.....38
Figure 7: Unlink Components – Link Deletion.....39
Figure 8: Sub-architecture Creation.....39
Figure 9: Join Components.....40
Figure 10: Extend Component.....41
Figure 11: Copy & Paste Component.....42
Figure 12: Operation Creation.....43
Figure 13: Operation Deletion.....43
Figure 14: Operation Attached To Component.....44
Figure 15: Operation Reset.....44
Figure 16: Move Operation.....45
Figure 17: Component Split.....46
Figure 18: Layer Change.....47
Figure 19: Move Component Across Layers.....48
Figure 20: Layer Info.....49
Figure 21: Save Architecture.....50
Figure 22: Load Architecture.....50
Figure 23: New Project.....51
Figure 24: Rapid Software Architecture Designer’s User Interface.....53
Figure 25: View’s internal architecture.....54
Figure 26: Workspace Bin Deletion.....55

Figure 27: Toolbar Buttons After Selection.....55
Figure 28: By Component Operation Display Mode.....56
Figure 29: Hierarchy Tree.....57
Figure 30: Component's Context Menu.....58
Figure 31: Workspace's Context Menu.....58
Figure 32: Component's Settings.....59
Figure 33: Custom Margin's Settings.....60
Figure 34: Settings of Description Extension.....60
Figure 35: Operation Settings.....61
Figure 36: Link Settings.....62
Figure 37: Super Mario Game Architecture.....67
Figure 38: Compiler's Architecture.....69
Figure 39: System Architecture's Initial Layer.....76
Figure 40: System Architecture's View Layer.....76
Figure 41: System Architecture's Blackboard Layer.....77

Chapter 1

Introduction

1.1. Background

Before proceeding to the main part of the thesis, we would like to present some basic notions in this section, for the readers that are not familiar with the area of software architecture design.

1.1.1. Components, Operations & Links

Software architecture constitutes one or more structures of the system, which includes software components, their in-between interactions and their properties, which must be always visible on the architecture [1]. A correctly built architecture is of vital importance in designing and constructing any complex software system, given that it provides portability, scalability, re-usability, high performance, interoperability and changeability. Intuitively, it acts as a connecting bridge between the requirements and the implementation [2].

The components of a software architecture represent the system's main computational parts (servers, clients, filters, etc) and data repositories (databases, Blackboards, etc) [1]. Additionally, a component refers to a functional role that directs the collection of related operational features to fulfill its purpose [3]. They appear as rounded rectangles, in the software architecture diagram and often there is an interaction among them [1].

The aforementioned interaction is called a link (or connector). According to Garlan, the links constitute the glue between the components, as they are responsible for the management of

the collaboration and communication processes [1]. On the architecture diagram, links are line segments that connect two components, which can be directed, bidirectional, or without any direction.

Another important part of software architecture is the operations that the designed system needs to perform. These operations signify requirements for the system's features, which can be standalone (orphan) or attached to a component [3]. On the software architecture diagram, they can be depicted with a rectangular shape inside the operations table.

1.1.2. Software Architecture Designing Process

The process of creating software architecture can be seen as a combination of both simple and complex design decisions. When converting requirements into a design, the software architect makes several decisions that when added together, produce the software architecture in its ultimate form [4]. More specifically, the architecture reveals how the designing system is broken down into components, including their operational features and how they are linked. These elements of design are key causes of mistakes when they are incompletely understood [5]. The architect can either start creating the components of the system or create the operations.

Apart from their creation, components could be deleted if their according roles should be removed. They may be joined if they have partial roles or split if they have excessive roles respectively. As for the communication among them, a link between them can be added. Finally, if a component’s role must be studied extensively, then a sub-architecture for this component can be inserted. Due to this process’s exploratory nature, many of the above actions can be repeated more than once, without any specific order until the architecture obtain its final form [3][4].

When the architect is finished with the component processing, they can proceed to configure the operations of the software architecture, if they have not already done so. The above notion

is applied, but different procedures are followed. Except for their creation and their deletion, operations can be attached to a component if they are correlated with its role. Otherwise, if one or more operations are already attached to a component and it is believed that they are not complying with its role they can be detached from it or moved to another component [3] [4].

The software architecture design process is considered complete when components and operations do not need to be altered further, and each operation has been set to a component.

1.1.3. Rapid Software Architecture Designing Process

In our tool, we chose to support and aid the acceleration of the software architecture design process, and therefore, it is very important to examine and analyze how this is achieved by reducing the excessive details on both the architecture diagram and its elements' data.

Concerning the creation software architecture diagram, when the user starts inserting components into the workspace, spends a lot of time deciding which is the best representing graphical element for each of the architecture's roles. During the editing of the architecture, one or more roles might change and the software architect must repeat the above process. Therefore, if all of the graphical elements that represent components are removed except for one simple rounded rectangle, saves the user a significant amount of time. The same notion applies also, to architecture's links and operations respectively.

Most software architecture design tools are extremely detailed requiring too much data from their users, to create or alter one or more architectural elements. Every time that a modification is made to the designed system, the data that concern the corresponding change must be altered accordingly. As a result, the user wastes valuable time for both creating and editing these elements. This issue can be solved by reducing the amount of information that the user is obligated to fill in.

1.2. Problem Definition & Objectives

Current software architecture designing tools are more detailed and use more information than it is required, which has a negative impact on both the architecture design process and the latter phases of development. Having users choose between multiple shapes to represent parts of the system is time-consuming and simultaneously hazardous for the architecture. Not only the abstraction that characterizes this process is compromised but also the visual element that is given to a part might not comply with its final role in the current or the next phase, resulting in its redesign. Moreover, it is unsuitable for the architect to put considerable effort into filling extra and unnecessary information, about the system, its components, links, and operations, which tend to make the already complex and demanding procedure of architecture designing even more difficult. Many tools also, support the insertion of blocks with URL links, notes, or even code on the workspace that not only is not terminal at this early stage of development but also can be confusing and overwhelming.

Our goal is the successful avoidance of all of the above issues, by creating a simple, rapid-by-design tool that assists the software architect to accomplish their goals faster and easier. First and foremost, our tool uses only re-sizable rounded rectangles to represent components, preventing the user to waste valuable time choosing the “right” visual element for this representation. Another equally important advantage of this type of representation is that the correct level of abstraction, in this phase of development, is assured. As for the information that is needed for creating a component, only its name must be filled and optionally a small description, which both can be altered later. The same notion is also applied to the operations and links. Therefore, our tool lets the user focus only on building the software architecture of the desired system and not caring about details that are excessive and overwhelming in this procedure.

1.3. Thesis Structure

In this segment, the structure of the rest of the thesis is analyzed. Firstly, some of the most popular similar software architecture designing tools are reviewed in chapter 2. In chapter 3, we study the system's architecture and its components. Moreover, in chapter 4 we analyze all the available actions, the user can make to construct an architecture. Each action is accompanied with by one figure for the reader to fully grasp their usage. Furthermore, in chapter 5 the user interface of our tool is examined as well as its configurations. Three different case studies are applied to our system and are thoroughly discussed in chapter 6. Finally, chapter 7 contains the conclusion of our work as well as a few suggestions for expanding further the capabilities of our tool.

Chapter 2

2. Related Work

In this chapter, we will go through a few tools that are relevant to our project. The software architecture design tools that are being presented briefly are divided into two categories: a) detailed specification tools and b) rapid design tools. Detailed specification tools necessitate extensive modeling of many operational elements of a system, which could occasionally result in the automated production of implementation parts. On the other hand, rapid design tools emphasize only on graphic and visual experimentation abstractly.

2.1. Detailed Specification Tools

In this section, we will examine the detailed specification tools.

StarUML

StarUML [6] is published by MKLabs and it is an open-source software engineering tool for system designing using the Unified Modeling Language, Systems Modeling Language and traditional modeling notations. Entity-relationship, data flow and flowchart diagrams can be generated with the assistance of this tool. Also, various third-party extensions and plugins can be added. For instance, code can be generated from the constructed diagrams via an add-on. Finally, StarUML is supported by macOS, Windows, and Linux operating systems.

Astah

Astah [7] is a UML modeling tool created by the Japanese company Change Vision. Apart from UML diagrams, ER diagrams, and mindmaps, it also supports use case, sequence, collaboration, deployment, and class diagrams, which makes the tool suitable for both industry and research usage. Also, the diagrams can be converted to Java source code files and vice versa. Multiple users can design and construct system architectures at the same time. Finally, many plugins are provided and can be installed on the tool.

Altova UModel

Altova UModel [8] is a UML tool created by Altova and allows the conversion of UML models in source code of C#, Java, C++, Visual Basic, ".NET", or in the project documentation. It supports also the reverse conversion from code to diagram. Its features also include UML2 diagram types and modeling of XML Schema in UML. Like the above design tools, UModel provides various types of diagrams that include class, component, sequence, behavioral, and many more. Moreover, it is suitable for embedded system engineers as it supports SySML [13]. Business process modeling notation [14] is also included in the tool. Finally, UModel can be installed as a plugin on IDEs like Eclipse [30] and Visual Studio [31].

Enterprise Architect

Sparx System's Enterprise Architect [9] is a visual modeling tool used for the design and construction of software systems. Via this tool, apart from ER, UML, component, and sequence diagrams, also model strategic and business diagrams can be created. Therefore, it can be used for both research and industry purposes. Finally, it supports the production of source code in languages like C#, C++, C, Java, etc., from the aforementioned diagrams.

Structurizr

Structurizr by Structurizr Limited [16], is a free-to-use software architecture design tool that requires the user to write code in Structurizr DSL [15], to be converted to a diagram.

Furthermore, the user can switch views between graph, diagram, UML, documentation, and the architecture decision records. The tool can be run on a cloud service or in a local host. The program can be run on the command line in combination with the Structurizr DSL, creating and exporting diagrams. Finally, after their creation, the diagrams can be exported in many various formats such as Plant UML [19], C4-Plant UML [20], Mermaid [17], Dot [18], and many more.

2.2. Rapid Designing Tools

In this segment, we will discuss the rapid design tools that are more close to our work, than the first category.

Archi

Archi [10] is a free open-source design tool made by Phillip Beauvoir and Jean-Baptiste Sarrodie. It is a cross-platform application and focuses on the creation of software architectures that do not require many pieces of information about the components and their connectors, from the architect. Its features also include the building of diagrams with its custom modeling language. The inverse conversion of diagrams to ArchiMate is also supported.

Gaphor

Gaphor [11] is a free, multi-platform, open-source modeling tool created by Arjan Molenaar and Dan Yeaw. The user can insert classes, connectors, state machines in the workspace, in order to construct a software architecture. Also, this tool provides Block Definition and Requirements diagrams for system designing. Finally, Gaphor supports many add-ons such as code generators, in which diagrams can be exported for documentation, and gives the user the ability to create his own custom plugins.

SAAM Tool

SAAM (Software Architecture Analysis Method) tool [12] is created to produce and evaluate software architecture designs. It features three modes of operation: a) build mode, b) scenario mode, and c) view mode. The build mode allows the user to develop the software architecture of a system that consists of components and connectors. In both of these architectural elements, a name and a description should be provided by the user. The scenario mode is used for creating scenarios and linking them with the related architectural elements. Finally, the view mode gives the user the ability to switch between multiple and different views of the designing architecture.

Diagrams.net

Diagrams.net [21] is an open-source, free-to-use graph modeling tool developed by the company JGraph Ltd. Not only Diagrams.net supports the creation of flowcharts and wireframes, but also provides UML, organizational, and network diagrams. These diagrams can also be stored in several cloud services such as Dropbox, Google Drive, OneDrive, and many more. Concerning the information that is required from the user, it is limited to inserting only the name of any architectural element. Finally, apart from the plugins that can be added to the tool, Diagrams.net can also be inserted into several other platforms as an add-on.

Lucidchart

Lucidchart [22] is a web-based design tool created by Lucid Software. Firstly, a variety of graphical elements are provided for the user to construct their software architecture. Multiple users can edit the architecture simultaneously by being co-authors. Furthermore, projects from different modeling tools such as Visio, Gliffy and OmniGraffle can be imported and edited. Finally, the current working project can be exported not only as a document in PDF, CSV, or Visio project but also as an image in the form of PNG and SVG.

Chapter 3

3. System Overview and Architecture

In this chapter the system's architecture as well as the implementation of three key components of the tool, will be discussed.

3.1. System's Architecture

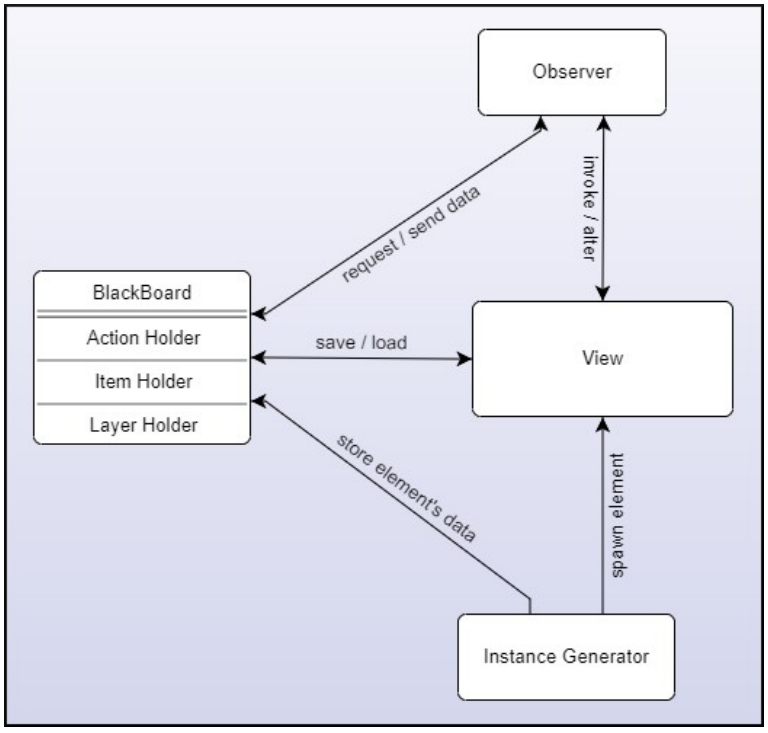

Figure 1: System's Architecture.

Our tool's architecture consists of four major components: the a) Instance Generator, b) Blackboard, c) Observer, and d) View. In this section, we will examine every component except the View, which will be analyzed in chapter 5.

3.2. Instance Generator

This component is responsible for handling and configuring the workspace and its items, which are all being produced with the assistance of the Go.js library [23].

First and foremost, it produces the Diagram object that will host both the components and links of the architecture. This production involves creating the object and configuring its necessary listeners, such as mouse button listeners. The panning ability and dotted grid are also been added to the workspace, through this process. Diagram's context menu is initialized and added on the right-click listener. After the construction and initialization of the diagram object, it is assigned to a member map of the Instance Creator, in which all diagrams are stored from all the layers of the architecture.

The Go.js library is used also to produce and initialize the components and their links. These architectural elements follow a specific template when they are created, which includes settings about their appearance and their listener's operations. In case the architect chooses to create a new component, then a JSON [25] containing its id, name, and position is added to the diagram's model with the assistance of an auxiliary function. Moreover, Instance Creator is responsible for not only creating but also deleting these elements and it provides methods that update the name and the description of the element when its details are edited. As we will see in chapter 5, the user can customize the appearance of all of the architectural items. Instance Creator provides specific methods to apply these changes to all the layers of the architecture at once and when the architect is adding a new item to the workspace its appearance is shaped, according to these modifications.

3.3. Blackboard

Our tool requires a shared space, to store workspace items, operations, layers, and user actions, that can be accessed for processing from the rest of the modules. As a solution to this

issue, the Blackboard pattern can be used [24]. The blackboard component consists of the Item, Layer, and Action Holder.

3.3.1. Item Holder

Every component, link & operation is stored in the Item Holder as an Item object. Item is a class containing all the necessary information about the architectural element that is referred to and its workspace representation. The Items are stored in an array inside the Item Holder. Apart from the basic insertion and deletion operations Item Holder provides item processing operations that can be called from the corresponding actions. For instance, if an operation is set to a component then the related function is called and updates internally both the component and operation Item, inside the blackboard. Finally, the Item Holder can be converted to a string for file saving.

3.3.2. Layer Holder

Every architectural layer is stored in the Layer Holder as a layer object. The layer class is similar to the Item class but it contains information related to the architectural layer. An Item Holder is in each Layer for complexity reasons. Similarly, it provides an additional and removal function to insert and remove layers from the layer array respectively. Moreover, Layer Holder provides a method called “Change Layer”, which is invoked every time the user wants to change the implementation layer via the Hierarchy Tree. As we will examine later on, it is also invoked on load and move to action. Finally, the Layer Holder can be converted to a string for the tool’s save and load function.

3.3.3. Action Holder

Our tool supports the Undo/Redo feature. More specifically, every action that does not concern architectural element configuration or side-tab configuration can be undone and redone. This feature is implemented with the help of three stacks. The command, the undo and redo stack. When the user changes something in the designer, then automatically their action is saved as a JSON in the command and the undo stack. The JSON contains the user's action, the inverse action, and both of the actions' data.

```
this.commands.push({
    action: actionExecuted,
    inverse: inverseAction,
    actionItems: {
        initialItem: originalItem,
        updatedItem: changedItem
    }
});
```

Figure 2: Actions' JSON.

When the architect invokes the undo function, the undo stack is popped and the popped command is pushed on the redo stack and the inverse process is followed when the redo function is invoked.

```
undo() {
    var lastUndoCommand = this.undoStack.pop();
    this.redoStack.push(lastUndoCommand);
    if (!lastUndoCommand)
        return;
    lastUndoCommand.inverse(lastUndoCommand.actionItems);
    return;
}
redo() {
    var lastRedoCommand = this.redoStack.pop();
    this.undoStack.push(lastRedoCommand);
    if (!lastRedoCommand)
        return;
    lastRedoCommand.action(lastRedoCommand.actionItems);
    return;
}
```

Figure 3: Undo and Redo methods.

3.4. Observer

This module is in charge of detecting every user input and invoking the related method. Firstly, when the user presses the right mouse button, the tool's custom context menu is displayed. As we will discuss in chapter 5, its options may vary according to where the event took place. For instance, if the user right-clicks a component, different context menu options will appear than if they right-click on a link. On the other hand, the single left mouse click is being used to alter the focus on the architectural elements and if combined with the left "Ctrl" button the user can select multiple elements at once. This focus is lost from all the selected elements when the workspace's background is clicked. Every time the selection is altered, the matching buttons on the toolbar are updated based on the number of focused objects.

Keyboard buttons are as much important to be detected as mouse button clicks. Firstly, as it is mentioned before the left "Ctrl" button plays a critical role when selecting architectural elements and must be detected at the same time with a mouse click. Also, when it is pressed in combination with the keys Z and Y, the undo and redo action is invoked respectively and when combined with the mouse wheel the user can adjust the tool's zoom. The Delete key is used when the architect chooses to delete the focused architectural elements. Finally, with the F11 key, the tool's full-screen mode can be enabled or disabled respectively.

Chapter 4

4. Software Architecture Designing Actions

In this chapter, we will examine the designing actions that our tool allows the architect to take, in order to construct a software architecture. Each action that will be described, if it is architecture altering, then it can be undone and redone via the undo and redo action respectively.

4.1. Component Actions

Below we will analyze the actions that can be performed on a component by the architect.

4.1.1 New & Delete

Upon component creation, a modal box is displayed for the user to fill optionally the component's name and description. Then the component is added to the workspace visually and simultaneously its information is stored on the current layer's item holder.

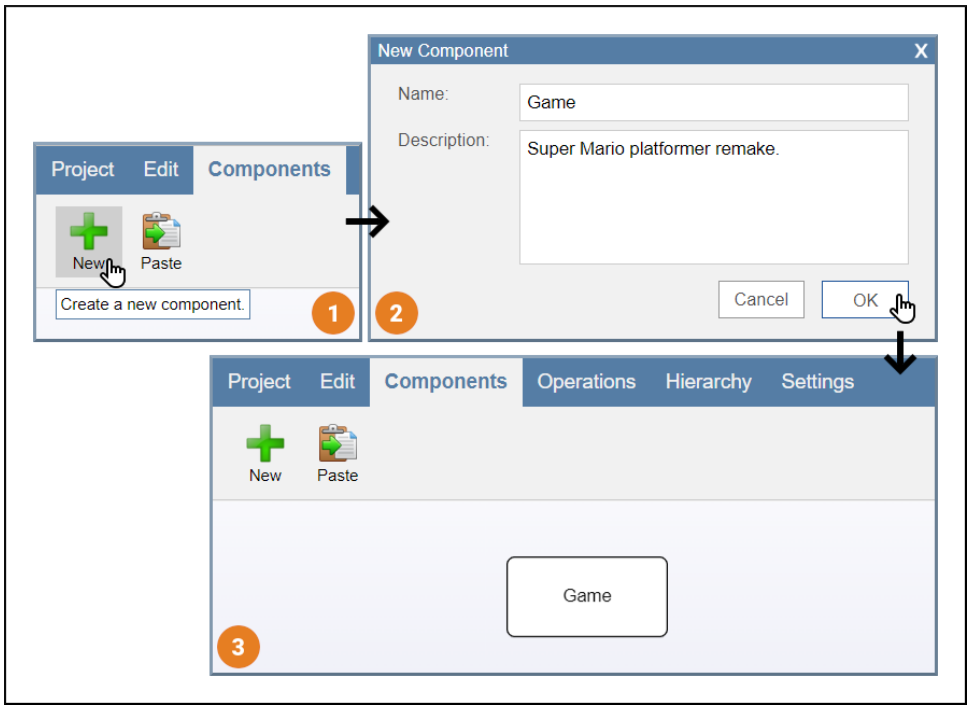

Figure 4: Component Creation.

If one or more components' role is no longer needed, the architect can remove them. When the user presses the delete button, a confirmation modal is displayed. If the user confirms their deletion, then they are removed visually from the workspace and the item holder.

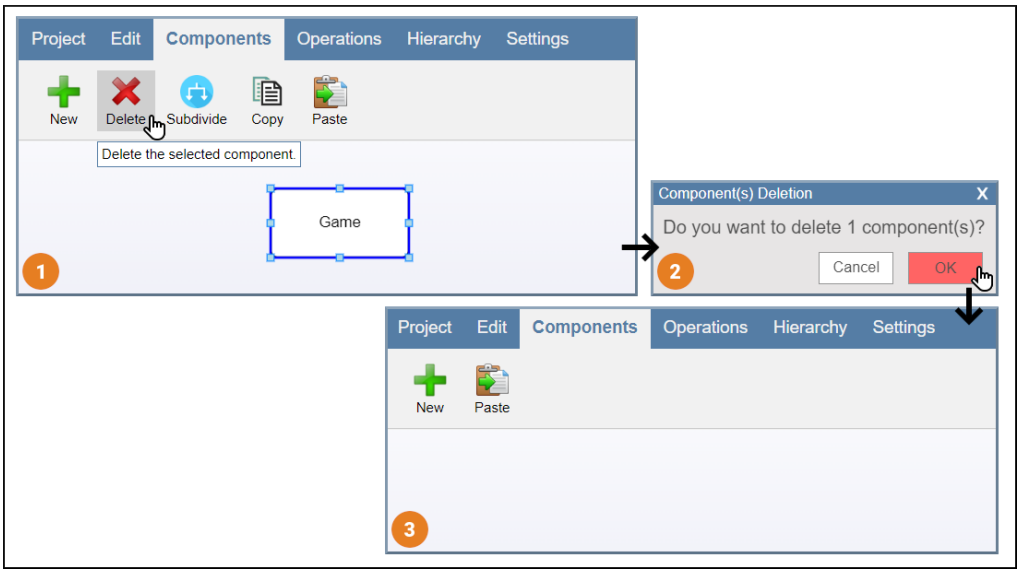

Figure 5: Component Deletion.

4.1.2 Link & Unlink

As we mentioned in the introduction, it is crucial for components to communicate, and to establish this communication, a link between them must be created. When two components are selected and the link button is pressed an input modal is displayed for the architect to fill the link's name and description. After that, the link is visually created and all of its information is stored on the item holder. Links can also be undirected, directed or bidirectional and if necessary this direction can be changed, via their context menu.

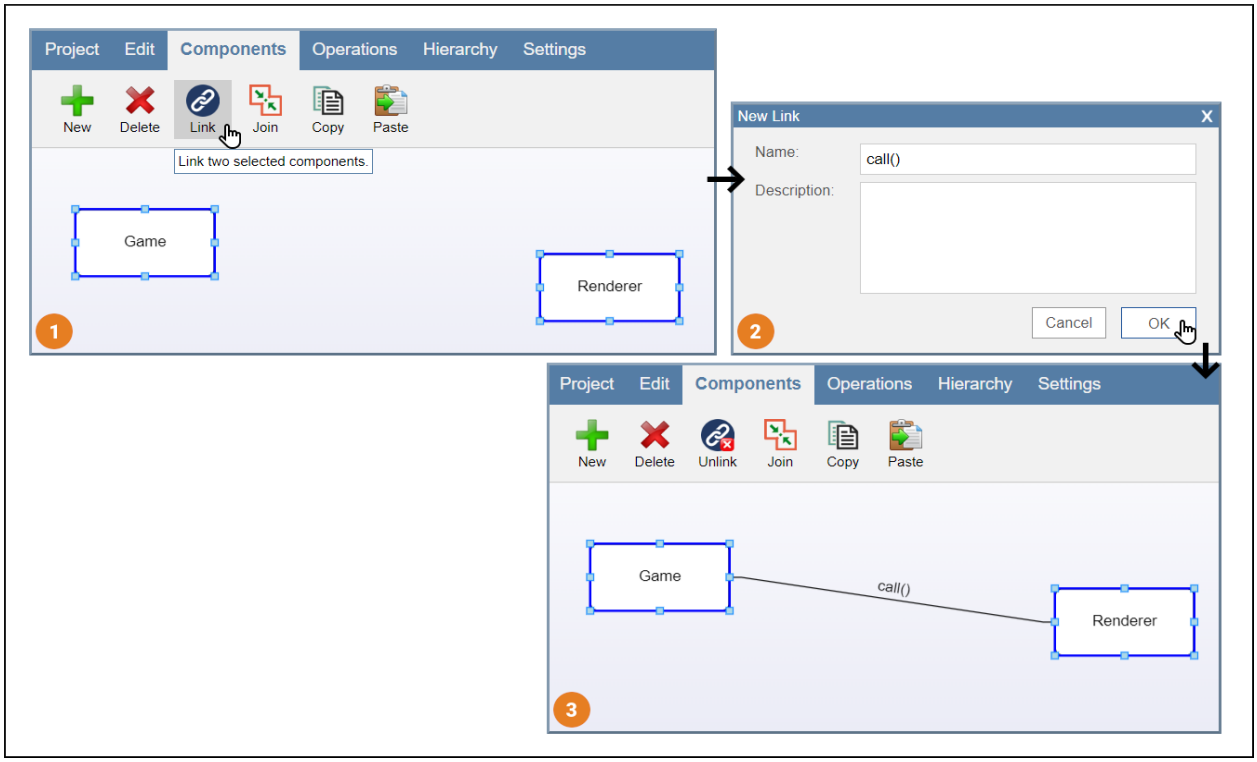

Figure 6: Link Creation.

If this connection turns out, it is no longer needed then it can be easily removed. More specifically, the link is deleted both graphically and from the item holder.

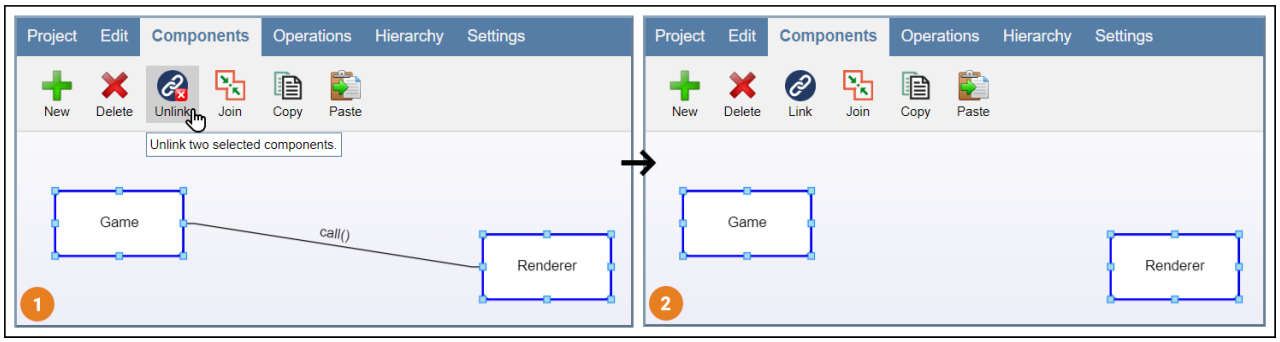

Figure 7: Unlink Components – Link Deletion.

4.1.3 Insert & Delete Component's Sub-architecture

A software architecture's component can be very complex to a degree that the user must create and design its sub-architecture. For the component's role to be studied further, a new layer of architecture with an empty workspace and operation table is being created. The layer is named after the component and the architect can switch to the newly created layer via the hierarchy tree.

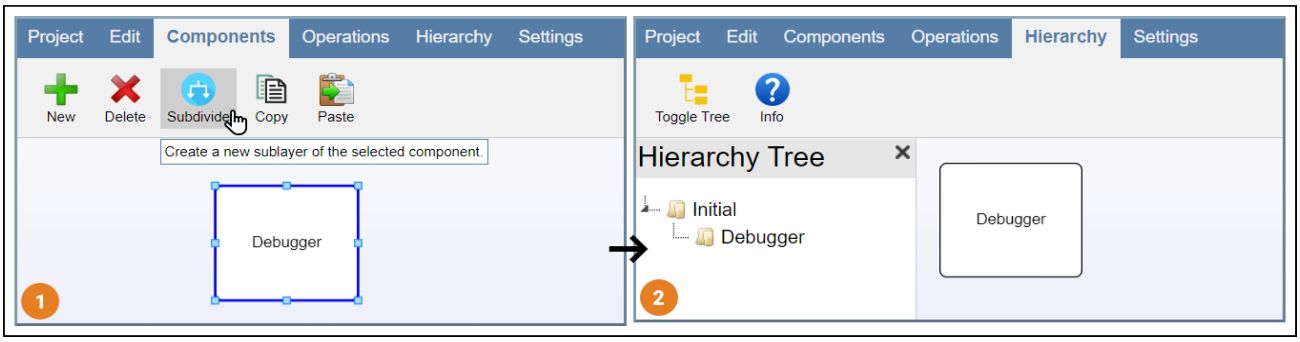

Figure 8: Sub-architecture Creation.

In case the architect realizes that the sub-architecture of a component is no longer needed due to the need for simplification, they can delete it with the inverse action.

4.1.4 Join Components

When two or more components are considered inadequate and have similar or partial functional roles then they can be joined [3]. If the components are connected between them the links are removed and all of their operations are transferred to the joined component. After the join, the component has a default name “Joined Item” that can be changed later by the architect.

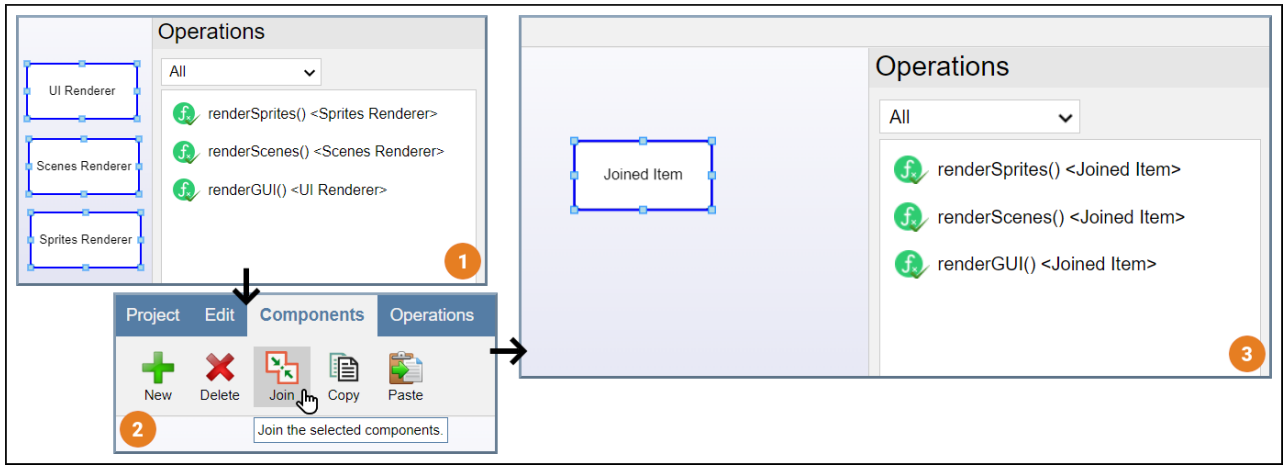

Figure 9: Join Components.

4.1.5 Extend & Collapse

Sub-architecture components are often required to appear in the upper layer. In case that happens the component can be extended with the names of its internal components. This way the user can constantly see a glimpse of the component’s sub-architecture while making other architectural decisions [27].

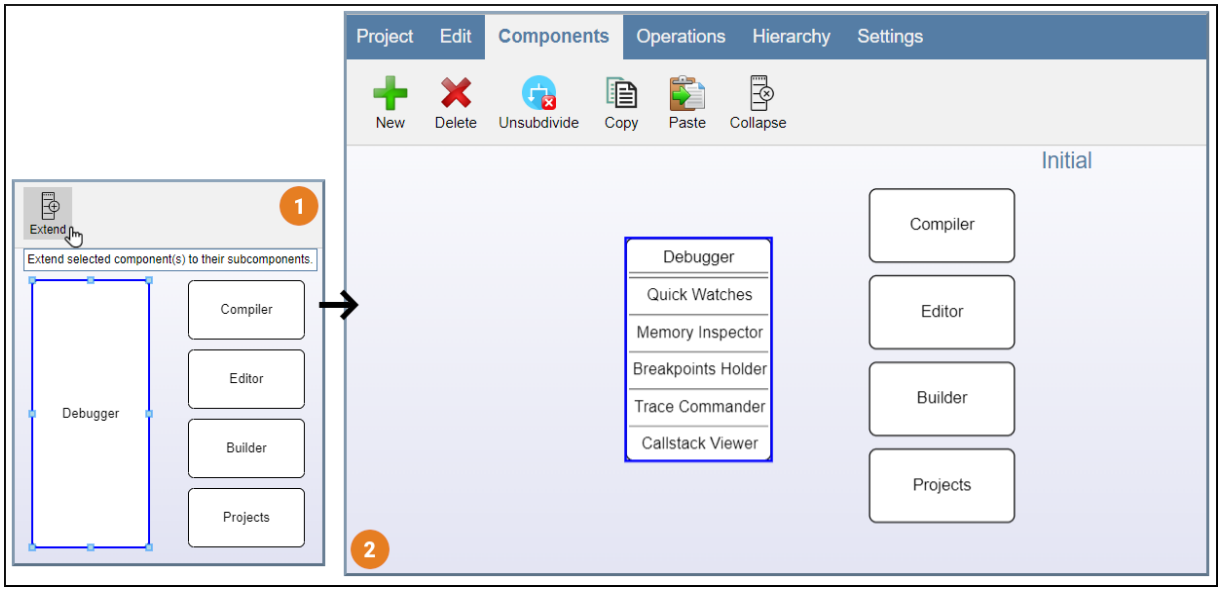

Figure 10: Extend Component.

If the sub-component appearance is no longer needed, the user can collapse the specific component to take its normal form.

4.1.6 Copy & Paste

When the architect wants to reuse one or more components in another instance of the tool, they can simply copy and paste them. Along with the components, the links between them, their operations, and their sub-architectures are also being transferred.

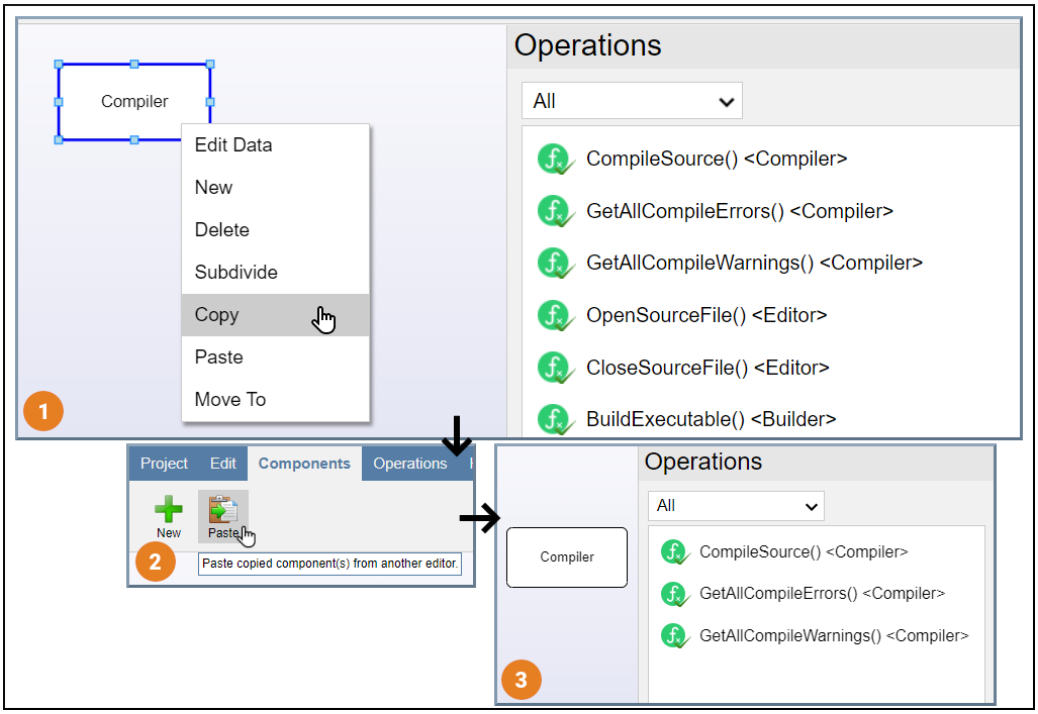

Figure 11: Copy & Paste Component.

4.2. Operation Actions

It is also of significant importance to analyze the actions that can be done upon an operation element, during the architecture design by the user.

4.2.1 Creation & Deletion

Similarly, with component and link creation, a modal is shown asking the user to fill in the name and an optional description for the newly created operation. After that, the operation rectangle is added to the operations table and its data on the item holder.

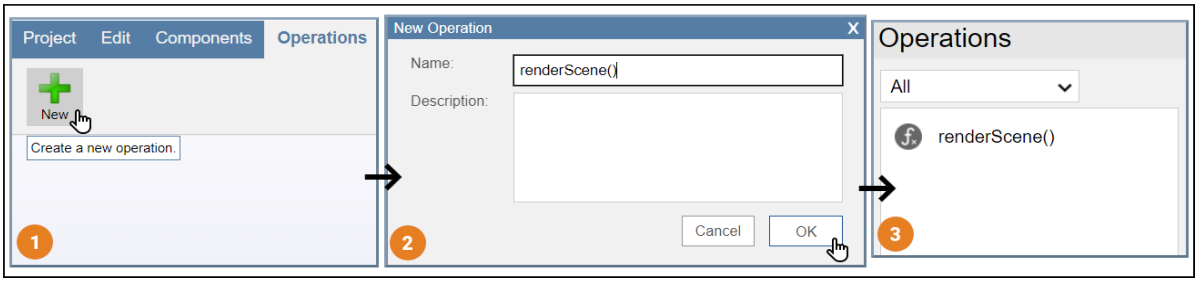

Figure 12: Operation Creation.

If the architect decides that one or more operations are unnecessary or inadequate, then they can select them and remove them.

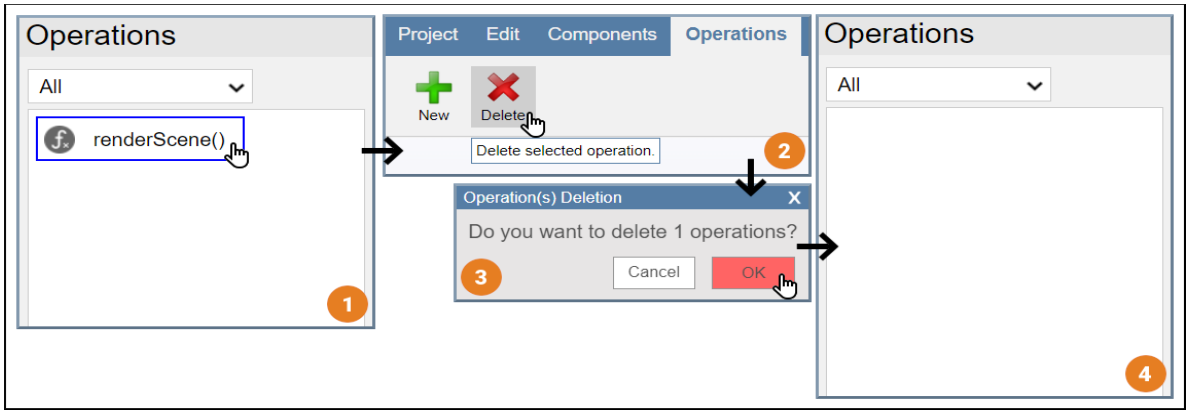

Figure 13: Operation Deletion.

4.2.2 Set & Reset

When it is determined, that an operation's functionality is matched with a component's role, then it can be attached to this component. This can be achieved, by simply dragging the operation over the component and dropping it. When that action is complete a message is popped and both the operation's owner and component's function list are updated in the item holder internally.

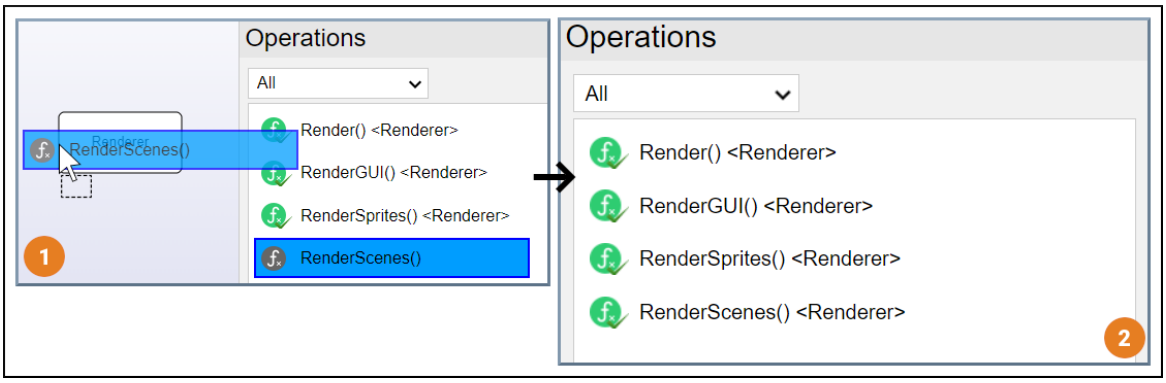

Figure 14: Operation Attached To Component.

A component's role might change and many operations that are attached can no longer comply with it. In this case, these operations should be reset from the component and become orphan operations. When these are selected, the user can simply click the reset button. All of the involved item's data are automatically updated in the item holder, by the completion of this action.

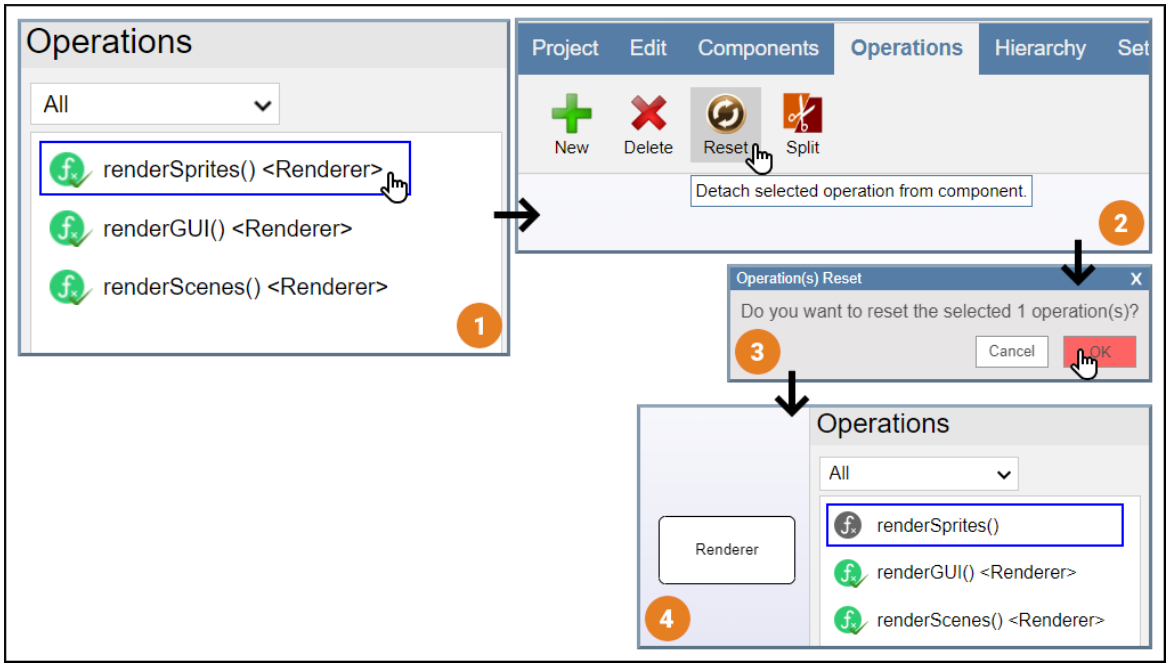

Figure 15: Operation Reset.

4.2.3 Move

One or more operations that are already assigned to a component might fit better with another component's role. The architect can move these operations by focusing on them, right click on one of them, and selecting the desired component to be transferred. As above, after this action, all of the involved items' information is updated on the items' blackboard.

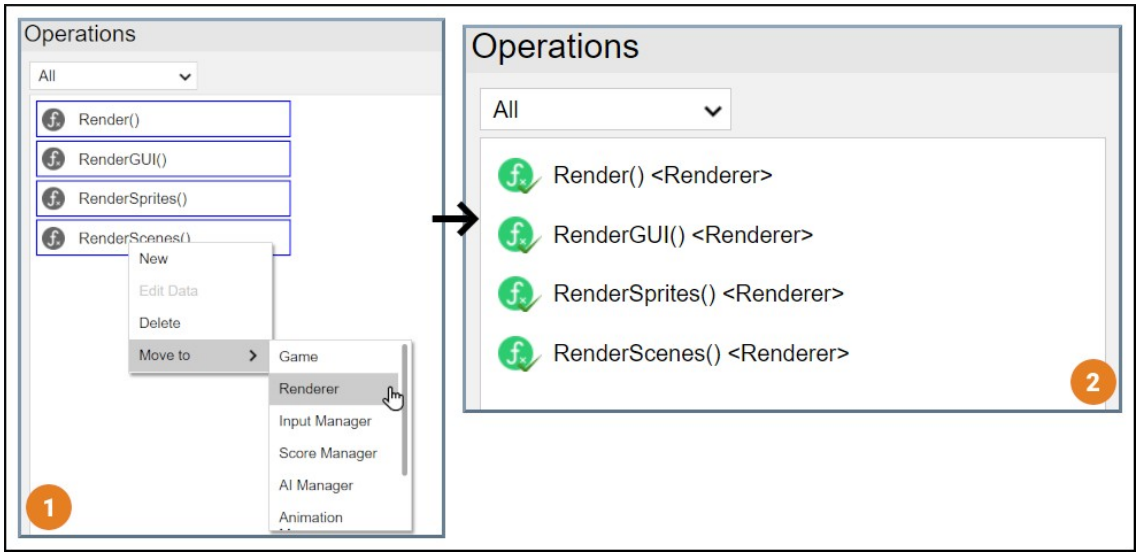

Figure 16: Move Operation.

4.2.4 Split

During the architecture design process, if there are one or more excessive operations on a component or if this component has become very complex owning too many operations, it should be split. Also, if several other operations from different components, do not fully correlate with their component's role, they can form another component via the split action [26]. After selecting the desired functions from the operation table and the user presses the split button, a new component is created with these functions attached. Automatically, these items' new information is changed on the item's blackboard.

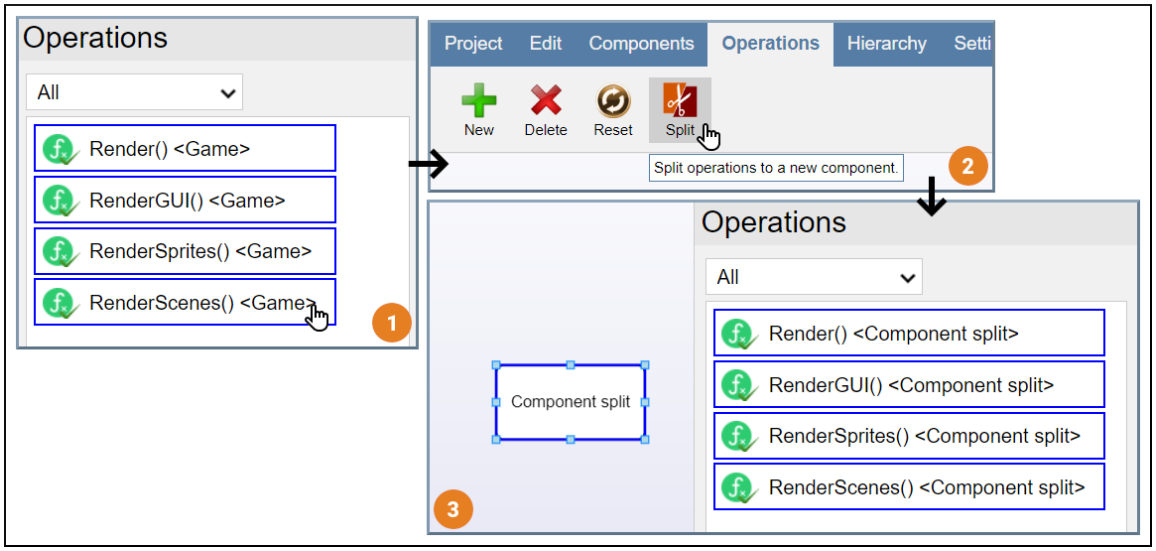

Figure 17: Component Split.

4.3. Layer Actions

This segment concerns the actions that can be made on an architectural layer and how these actions affect the layer holder and layer hierarchy.

4.3.1 Creation & Deletion

As it is described before, both of the creation and deletion of layers are depended on the components of the architecture. If a role should be thoroughly studied then it should be subdivided, producing a new layer. On the other hand, if the architect determines that this role should no longer have an implementation, then the component's layer is deleted. In both cases, the layer holder is updated accordingly.

4.3.2 Layer Change

The current layer can be switched, using the hierarchy tree. Both the operation table and workspace are altered according to the changed layer functions, components, and links. After this switch, any item's data adjustments are updated on the changed layer's item holder.

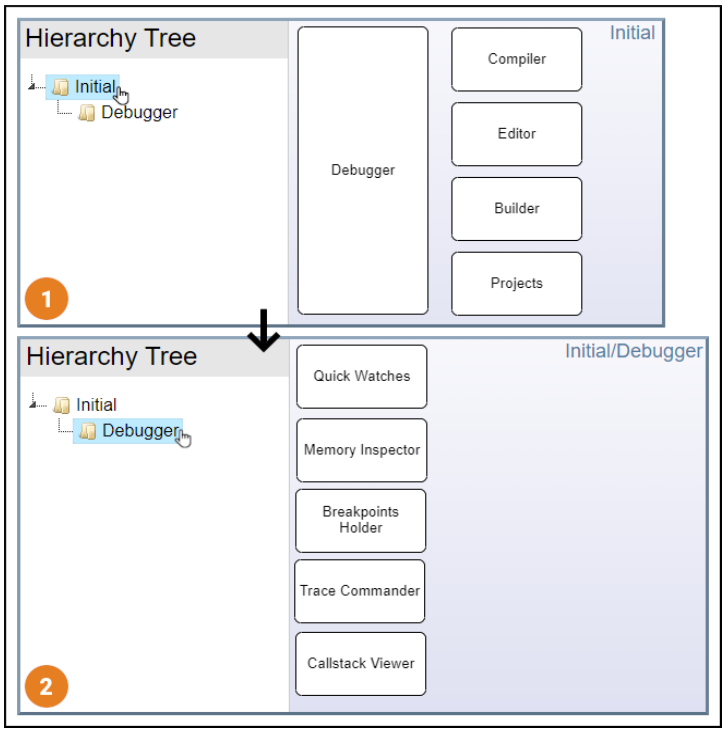

Figure 18: Layer Change.

4.3.3 Move Components Across Layers

If necessary, the architect is allowed to move one or more components to another layer. This can be accomplished by simply selecting the desired components and pressing the "Move To" button. The links, operations, and sub-architectures of the transferred components are also moved. The layer and item holders are updated accordingly.

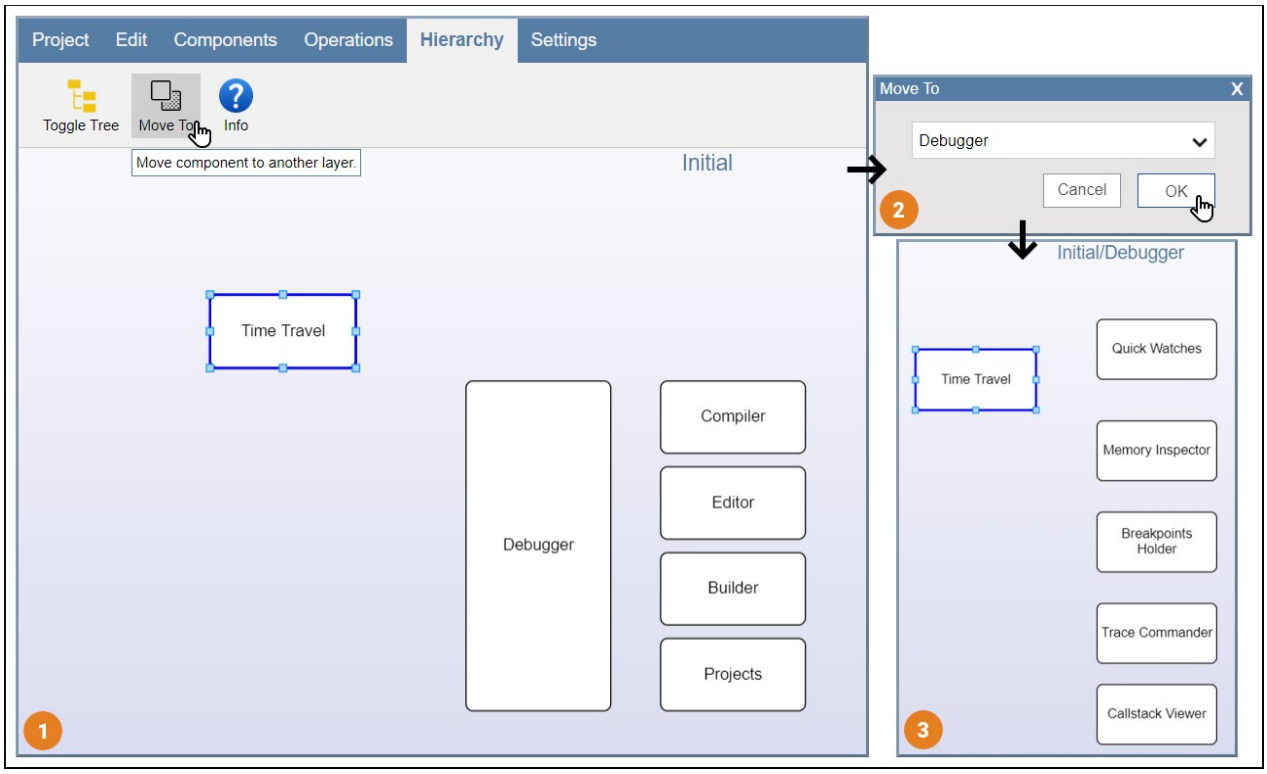

Figure 19: Move Component Across Layers.

4.3.4 Layer Info

A layer report is necessary to inform the user about: a) how many operations are orphan, b) how many components do not have any operation attached to them, c) which component has the maximum and d) which has the minimum amount of operations. In this manner, assistance is provided to the architect by the system, facilitating their decision-making and aiding the planning of their next steps. Also, next to each of the rows, there are hint buttons, and when pressed the corresponding element is focused. Every time that a change is made to the software architecture that concerns the aforementioned information, the modal's content is altered accordingly.

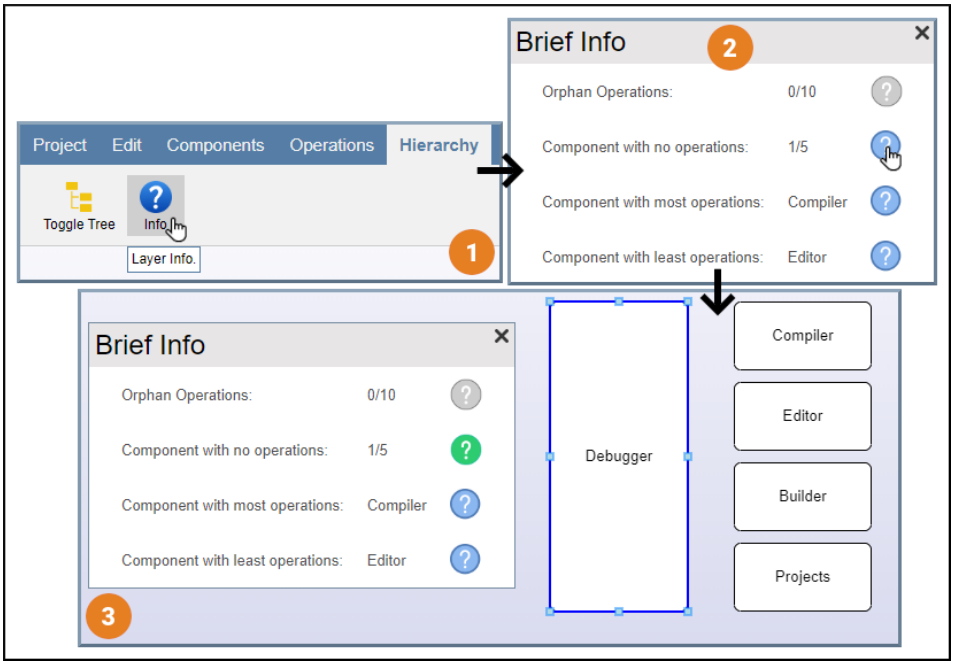

Figure 20: Layer Info.

4.4. Project Actions

This section concerns project manipulation operations. The architect can save/load their work or create a new project.

4.4.1 Save & Load Project

It is of vital importance, that the user after fully or partially designing a software architecture, is allowed to save it. After giving the architecture a name, the file is downloaded in JSON form, with the suffix “prj”. It contains all the necessary information (layer & item holders) to reconstruct the saved architecture by loading it into the system.

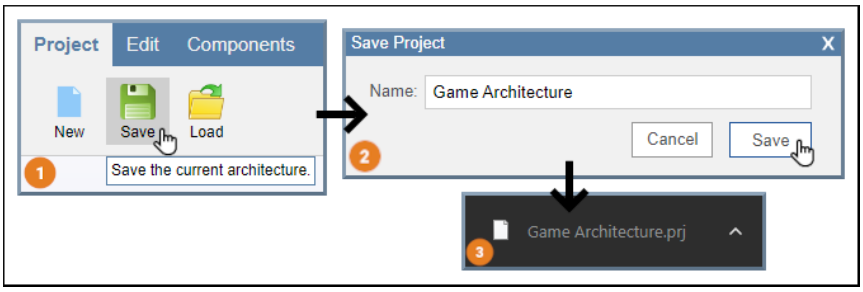

Figure 21: Save Architecture.

Similarly, when the architect is willing to continue the design of a saved architecture, they can simply load it into the tool.

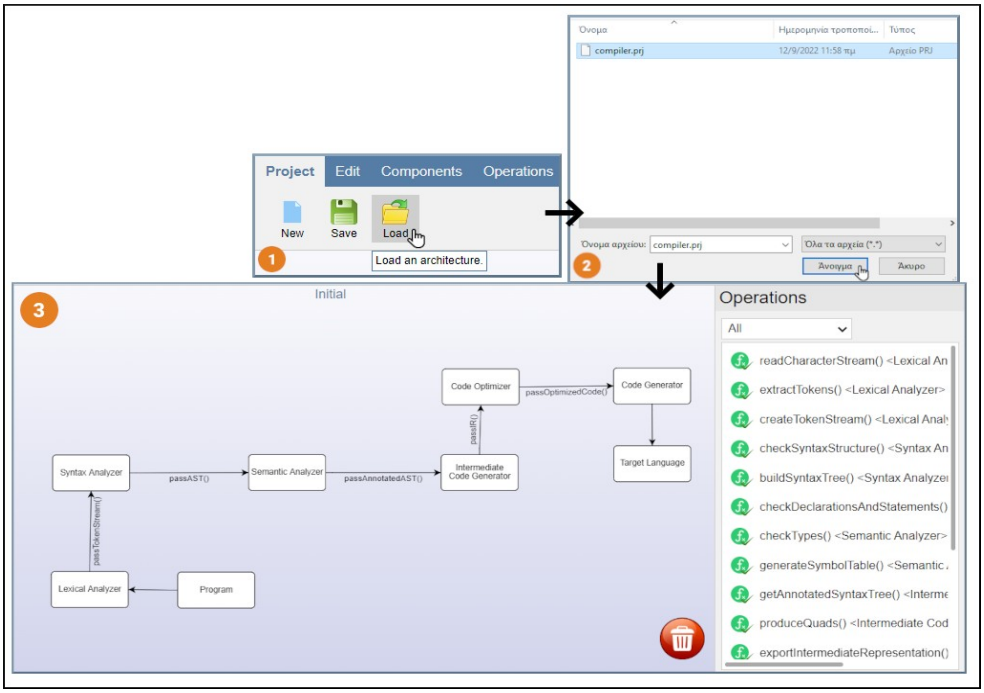

Figure 22: Load Architecture.

4.4.2 New Project

The architect after finishing an architecture can create a new one from scratch with new project action. Optionally the user can save the currently designed architecture before removing it from the tool, preventing it from being lost.

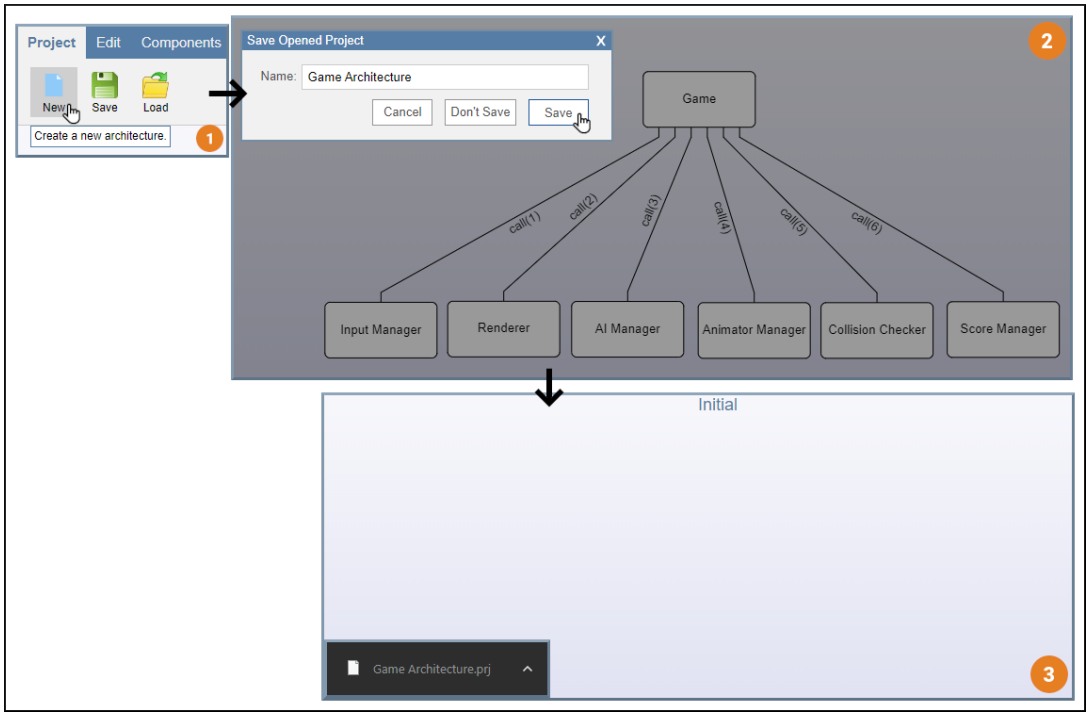

Figure 23: New Project.

Chapter 5

5. Software Architecture Visualization & Configuration

In this chapter, we will discuss the View component of the tool's architecture as well as the configuration settings that the architect is allowed to alter.

5.1. User Interface

Figure 24 shows our tool's user interface. On the top of the figure, the designer's toolbar can be seen and on the right, the operation table is located, with the working architectural layer's operations inside of it. The operation display mode can be switched by the selecting field on the right tab. The components and links of the designed architecture are inside the workspace which has a "gradient" background.

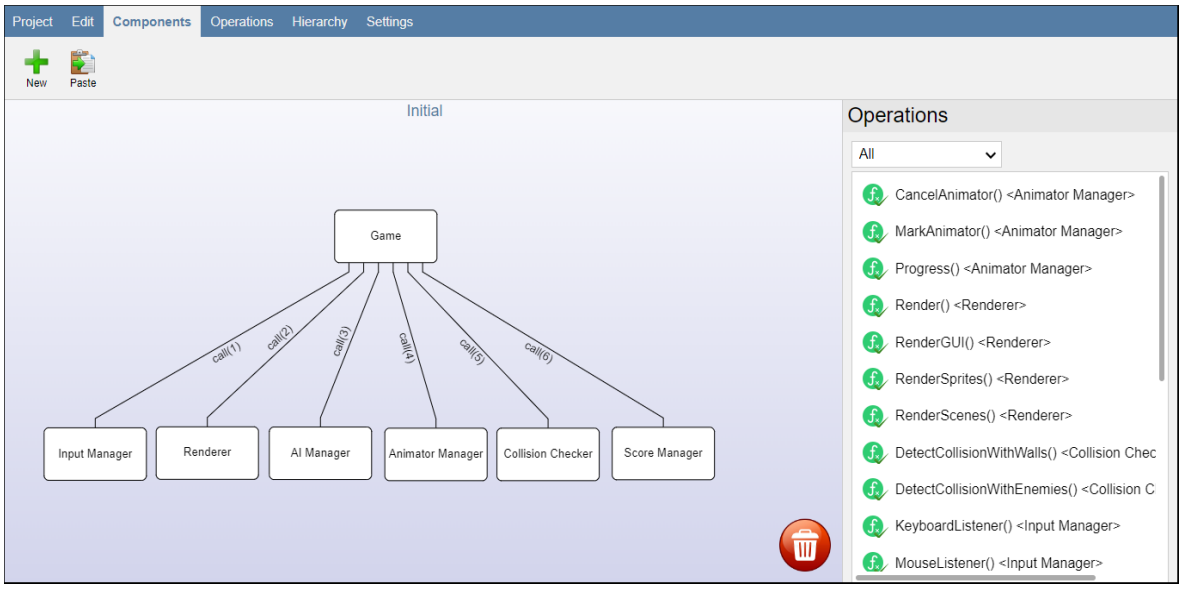

Figure 24: Rapid Software Architecture Designer's User Interface.

5.2. Implementation

In this section, we will examine the implementation of the workspace, toolbar, operations table, and layer hierarchy. These components constitute the internal implementation of the View component.

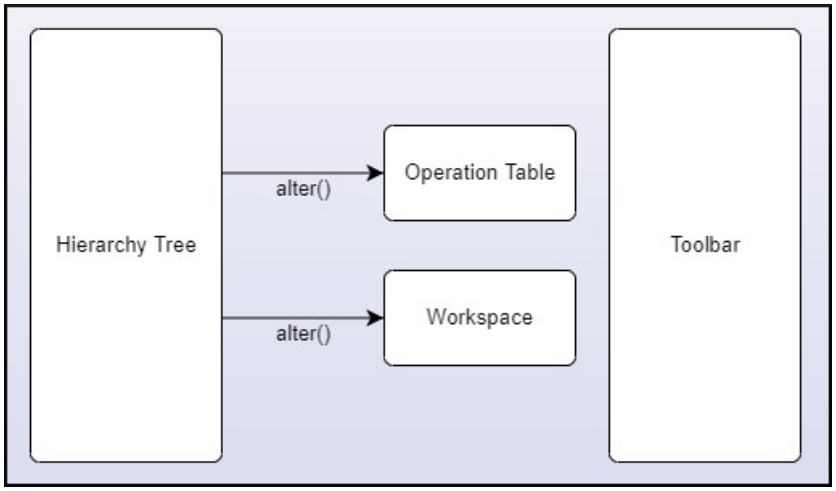

Figure 25: View's internal architecture.

5.2.1 Workspace

This area contains the software architecture's components and links. The user can move and re-size components inside the workspace and if one or more components are linked to them, then their connector's position is changed accordingly. If the user tries to move one or more components out of workspace bounds, then the view is scrolled in that specific direction. Moreover, the architect can zoom in or out on the designed architecture or move the whole architecture using panning. Multiple links and components can be selected via the selecting rectangle. On the bottom right of the workspace, the bin is used for component deletion by the architect.

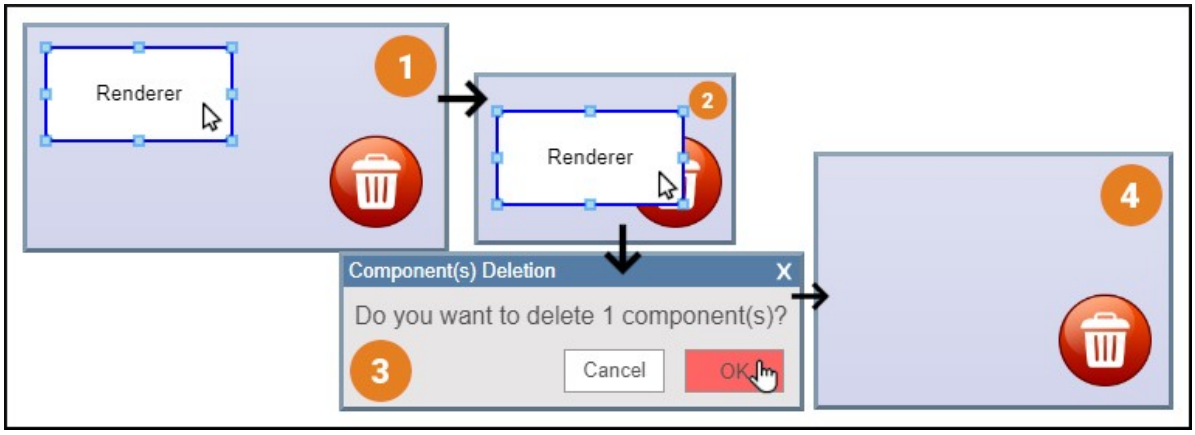

Figure 26: Workspace Bin Deletion.

5.2.2 Toolbar

The architecture designing functions that have been discussed in Chapter 4 can be executed using the application's toolbar buttons. These buttons are divided into tabs based on their category. A specific precondition must be fulfilled for some functions to be executed. Therefore, the related buttons are hidden when that precondition is false. For instance, the delete button on the component tab is hidden until the user selects one or more components. This can be achieved with the assistance of the observer component that we have examined in chapter 3. If a button is hovered by the mouse pointer, then a brief description of its role will appear.

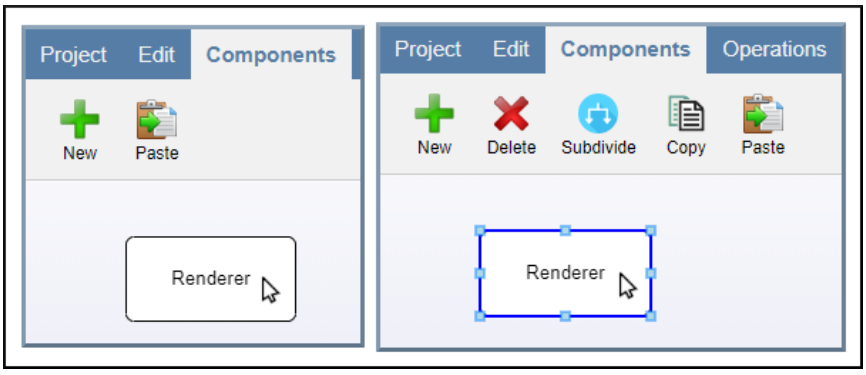

Figure 27: Toolbar Buttons After Selection.

5.2.3 Operations' Table

As it is mentioned in the previous chapters, the operation table of the tool contains the list of all of the operations on the currently working layer. It provides two modes of viewing the operations. The first mode is called “All” in which all the operations are visible. On the other hand, when the mode “By Component” is selected, the architect can only see the operations of currently selected components. Moreover, when the layer of the architecture is changed, the operation table is refreshed and contains the currently selected layer’s operations.

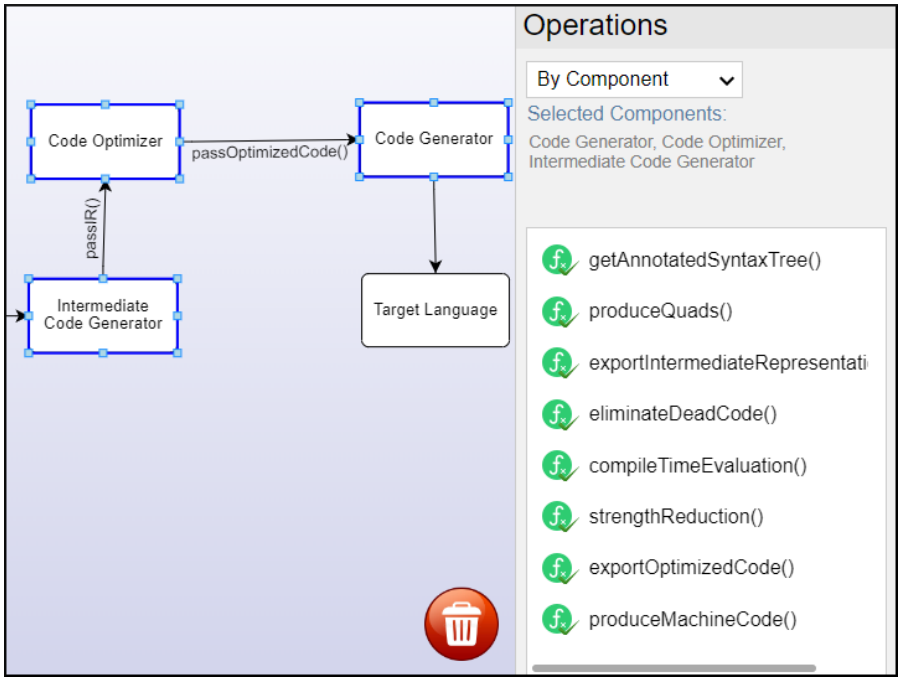

Figure 28: By Component Operation Display Mode.

If the user hovers over any operation, then a “tooltip” [29] appears with the operation’s description. Operations can also be dragged as shown in figure 14 in the 4.2 section, which allows the architect to easily drag them and drop them on a component to attach them. Finally, the right tab can be extended by dragging its left edge towards the left or right, and depending on the direction, its width is altered accordingly.

5.2.4 Hierarchy Tree

On the focus of the hierarchy tab, the hierarchy tree appears. More specifically, the hierarchy tree is a list in the form of a tree view, containing all layers of architecture with hierarchical order. It is constructed with the help of the JSTree library [28]. When a layer is clicked, the workspace and the operation table are changed accordingly to this selection.

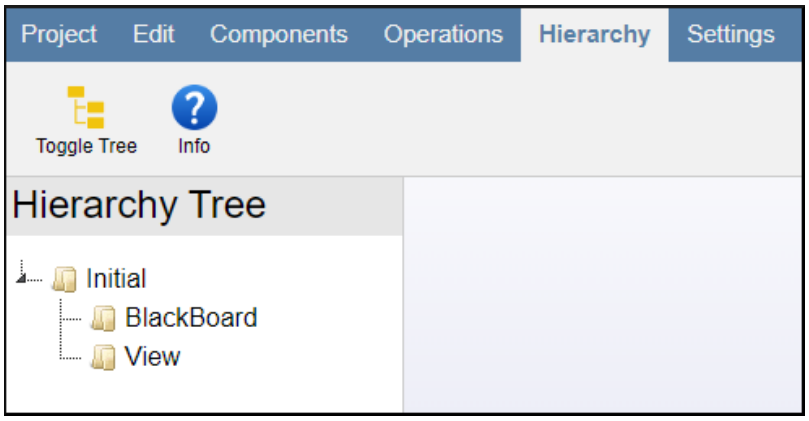

Figure 29: Hierarchy Tree.

5.3. Context Menus

The rapid architecture designer provides 4 types of context menus: a) component, b) link, c) operation and d) workspace context menu. Firstly, a component's context menu contains the "edit data" option which allows the user to edit the name and description of the component, as well as all of the component tab's currently visible button's actions. The same notion applies to the operation context menu.

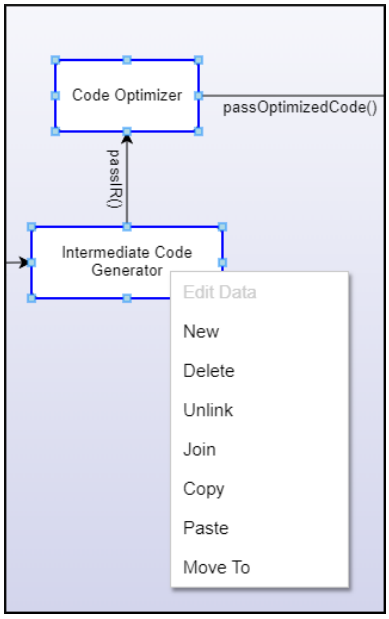

Figure 30: Component's Context Menu.

On the other hand, links' context menu has only the "edit data" and "choose direction" option, because their creation and deletion depend on the components. The workspace context menu provides the architect with the choice to create a new component or operation as well as to add a grid and grid snapping ability to the workspace. Also, the user can enable the tool's fullscreen mode. Finally, via the workspace menu, one or more components can be pasted and if an action is made it can be undone or redone.

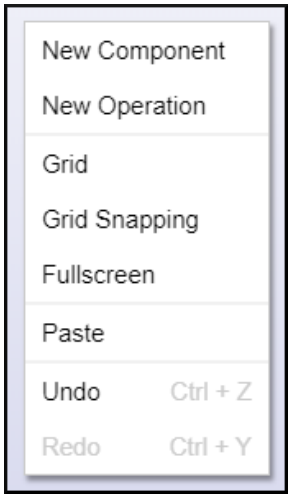

Figure 31: Workspace's Context Menu.

5.4. Configurations

The appearance of all tool's architectural elements can be altered massively all at once, for every layer of the architecture. In all three elements, the architect can change the font's size, family, color, background color, weight, style, and decoration. The user has also the option to reset to default configurations. Finally, all of the configurations can be stored or can be imported from a configuration file.

5.4.1 Component Settings

First and foremost, the component's color can be altered as well as the appearance of its border. The border width can be adjusted with the assistance of a slide bar. Both the focused and unfocused border colors can be changed. As we have examined in chapter 4 components can be extended. The letter and background color of these sub-components can be modified.

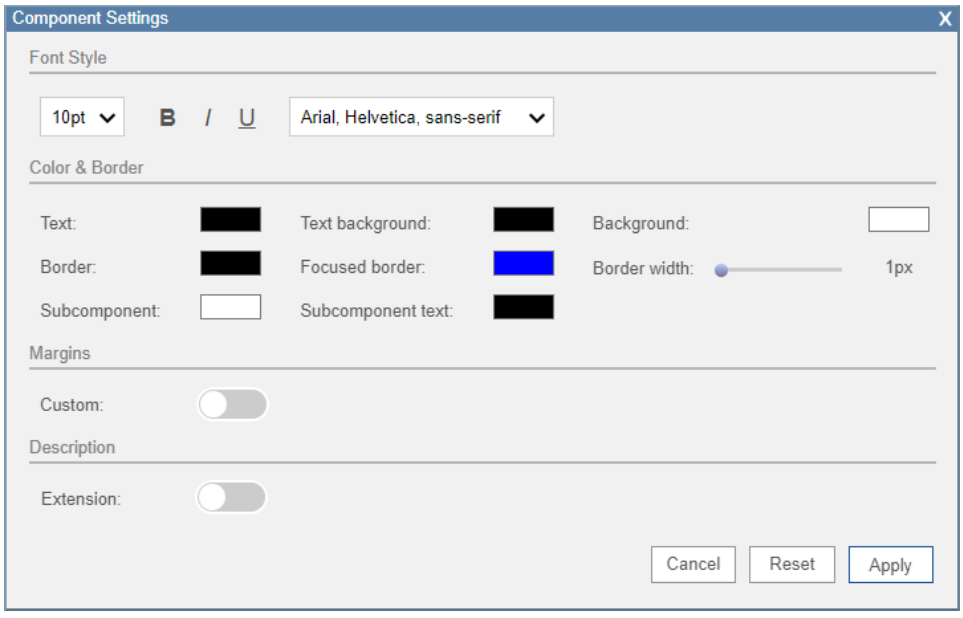

Figure 32: Component's Settings.

The horizontal and vertical distance between the component's text and its border can be also adjusted by the architect. When the margin switch is enabled, both the horizontal and vertical distance is 19px, which can be altered again via slide bars.

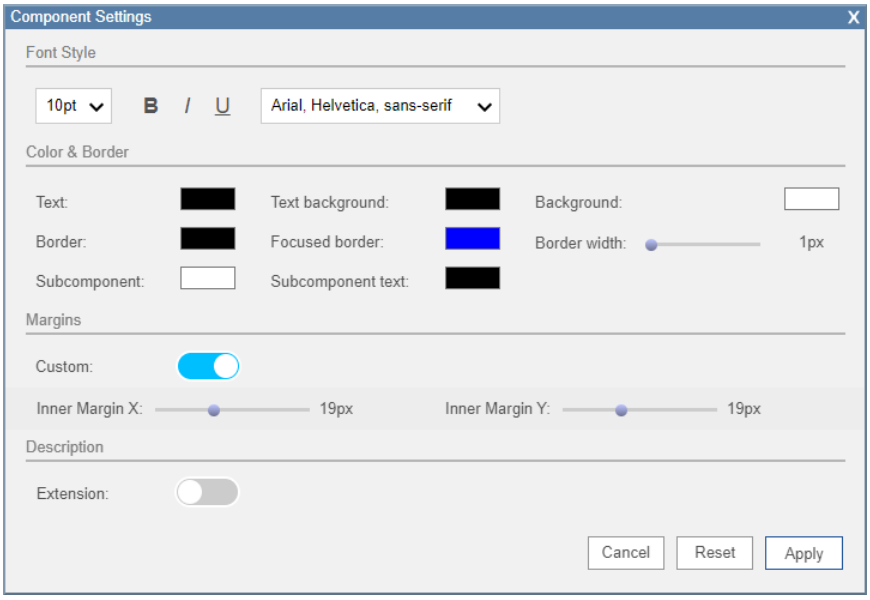

Figure 33: Custom Margin's Settings.

The component can also be extended in a way that the description is visible at all times. The appearance is the same when it's extended with sub-components except for the extended part's content.

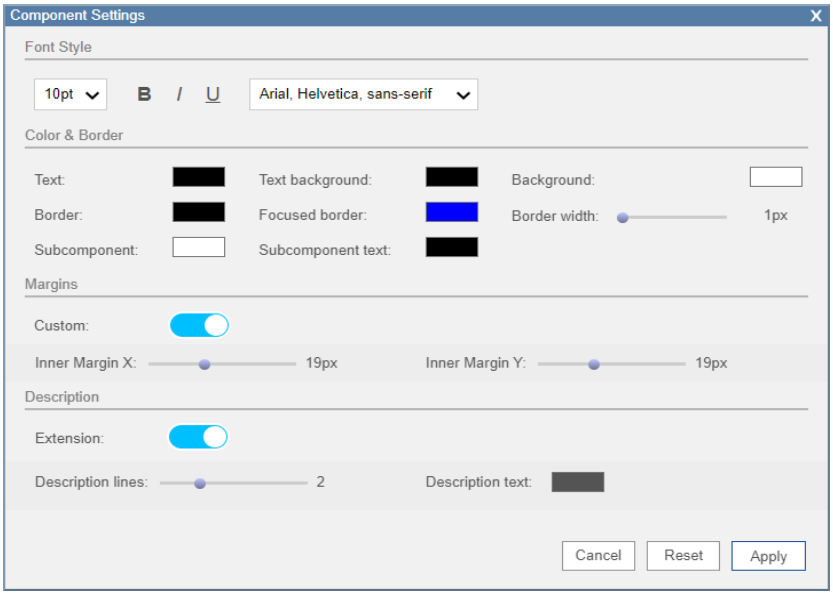

Figure 34: Settings of Description Extension.

5.4.2 Operation Settings

Firstly, the operation's background can be modified by the user. Similarly to the component, both the focused and unfocused border's color of the operation can be altered. Finally, when an operation is being dragged, its color changes to blue. This can be altered by the user via these settings.

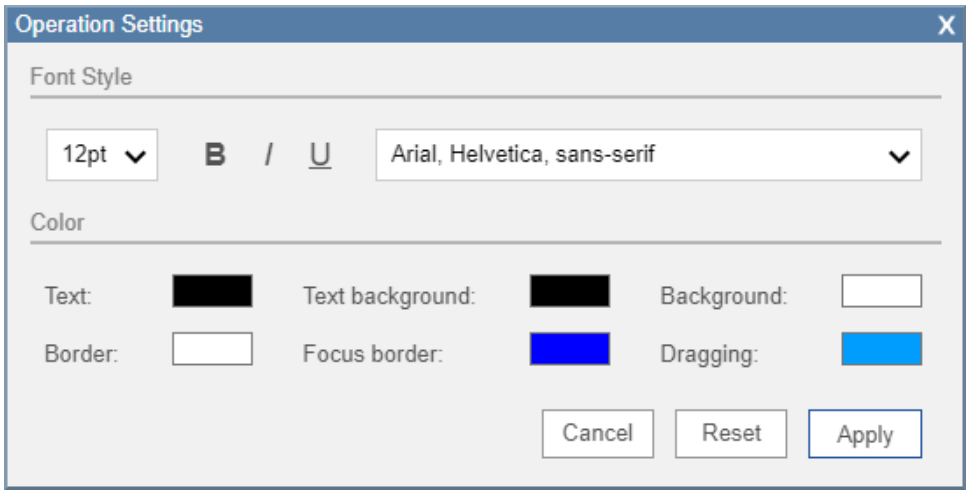

Figure 35: Operation Settings.

5.4.3 Link Settings

Apart from its label, which contains its name, both link's line segment and arrow colors can be modified.

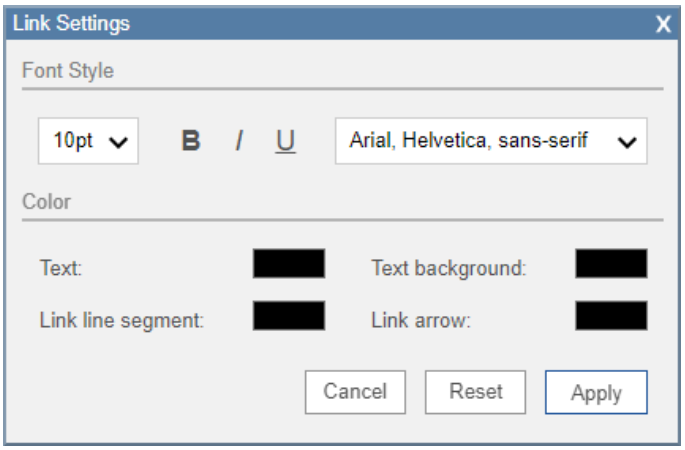

Figure 36: Link Settings.

Chapter 6

6. Case Study

In order for our rapid software architecture designer to be tested, in the following subsections we applied three different case studies. Firstly, we will build a game engine architecture. After that a compiler architecture will be constructed and finally the tool's system's architecture itself will be designed. In these case studies, we will examine their functional roles and their abstract operations.

6.1. Super Mario Game Engine's Architecture

In this segment, the super Mario game engine's software architecture will be designed and studied.

6.1.1. Components

The components which represent the functional roles of the super Mario game engine's architecture are mentioned below:

- **Game** constitutes the main component of the game engine, which communicates with the rest of the components. More specifically, invokes methods provided by Input Manager, Renderer, AI Manager, Animation Manager, Collision Checker and Score Manager.
- **Input Manager** detects keyboard and mouse inputs made by the user.
- **Renderer** handles the rendering of GUI, Scenes and Sprites.
- **AI Manager** is responsible for configuring and handling AI algorithms for the various enemies.

- **Animator Manager's** role includes animators' storage and handling for all the entities that require one.
- **Collision Checker** detects both non playable and playable characters' collisions with the game's environment. It is also responsible for item collision such as coins and mushrooms.
- **Score Manager** collects all the score points which are gained by obtaining coins, mushrooms or by killing enemies.

6.1.2. Operations

The functions that the game engine requires to perform are the following:

- **ExecuteGameLoop()** (*attached to the Game*) is invoked by the main operation and constitutes the essential loop that is executed by almost all categories of action video games [32]. It is responsible for invoking operations for rendering, handling user input, progressing the animations, configuring non playable characters' behavior, and checking for collisions.
- **Main()** (*attached to Game*) is the main operation of the game engine which after all the necessary initializations are made, invokes the game loop.
- **KeyboardListener()** (*attached to the Input Manager*) is responsible for detecting keyboard input.
- **MouseListener()** (*attached to the Input Manager*) is responsible for detecting mouse button's input.
- **Render()** (*attached to the Renderer*) is the main rendering operation which invokes the methods: a) RenderGUI(), b) RenderScenes() and c) RenderSprites().
- **RenderGUI()** (*attached to the Renderer*) renders the graphical user interface.
- **RenderScenes()** (*attached to the Renderer*) renders the game's scenes.
- **RenderSprites()** (*attached to the Renderer*) renders the item's, enemies' and Mario's sprites.

- **ApplyConceptualAI()** (*attached to the AI Manager*) applies artificial intelligence algorithms for the different types of enemies' behaviors.
- **CancelAnimator()** (*attached to the Animator Manager*) cancels a specific animator.
- **Mark()** (*attached to the Animator Manager*) marks an animator as running (currently executing) or suspended (stopped).
- **Progress()** (*attached to the Animator Manager*) progresses a specific Animator.
- **RegisterAnimator()** (*attached to the Animator Manager*) registers a specific Animator.
- **DetectCollisionWithWalls()** (attached to *the* Collision Checker) detects the collision between an item and a wall.
- **DetectCollisionWithEnemies()** (*attached to the Collision Checker*) detects the collision between Mario and the enemies.
- **DetectCollisionWithPowerUp()** (*attached to the Collision Checker*) detects the collision between Mario and a power up.
- **DetectCollisionWithCoins()** (*attached to the Collision Checker*) detects the collision between Mario and a coin.
- **CalculateCoinPoints()** (*attached to the Score Manager*) calculates the collected coins' points and adds them to the final score.
- **CalculateKillPoints()** (*attached to the Score Manager)* calculates Mario's kill points and adds them to the final score.
- **CalculatePowerPoints()** (*attached to the Score Manager*) calculates power-ups' points and adds them to the final score.

6.1.3. Final Result

Below, we can see the fully constructed super Mario game architecture with its components, operations and links.

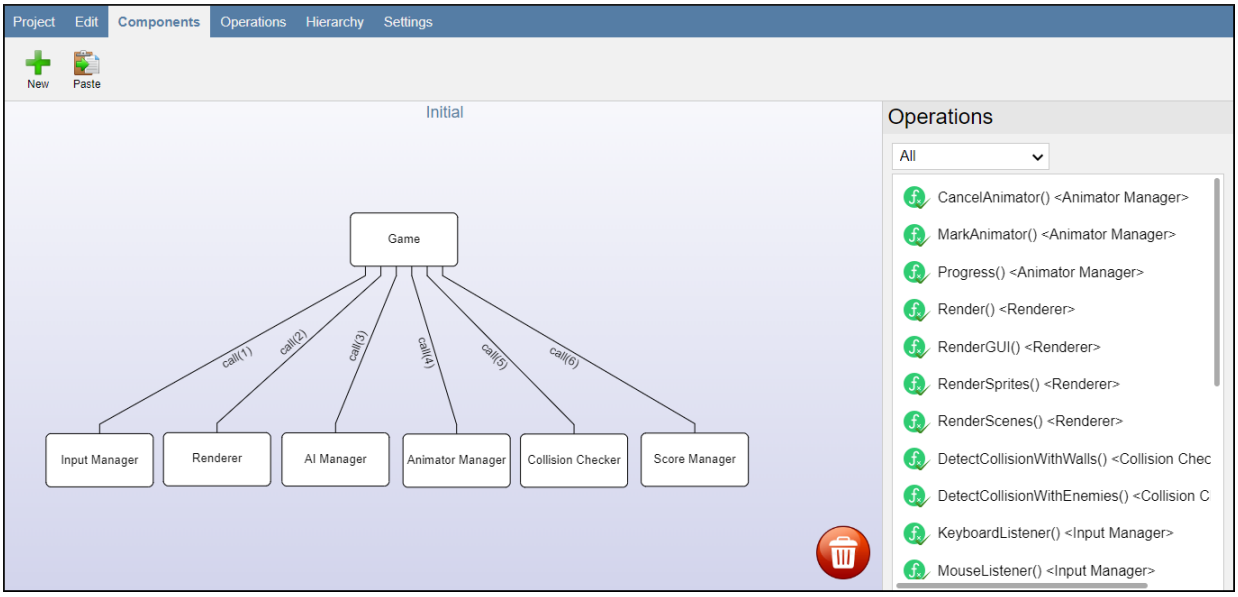

Figure 37: Super Mario Game Architecture.

6.2. Compiler's Architecture

Our next case study concerns the creation of a compiler's software architecture. As above we will examine first its components and then its operations.

6.2.1. Components

The compiler case study's components are the following:

- **Program** represents user's input program.
- **Lexical Analyzer** reads user's program and extracts its tokens, creating a stream. If there is an unknown character displays the related error message.
- **Syntax Analyzer** checks if there is a syntax error and constructs the syntax tree.
- **Semantic Analyzer** checks variables' types, user's declarations and statements. When this process is finished without errors, it produces the symbol table.
- **Intermediate Code Generator** produces the intermediate code representation quads and passes them to code optimizer.

- **Code Optimizer** improves the intermediate code by optimizing it and as a result consumes fewer resources. The optimized code is sent to Code Generator.
- **Code Generator** produces and exports the target code.
- **Target Code** component represents the target code.

6.2.2. Operations

The compiler case study's operations are the following:

- **ExtractTokens()** (*attached to the Lexical Analyzer*) assorts the scanned code to tokens (keywords, constants, operators etc). If an undefined token is detected an error is thrown.
- **ReadCharacterStream()** (*attached to the Lexical Analyzer*) scans the user's code.
- **CreateTokenStream()** (*attached to the Lexical Analyzer*) takes the stored tokens and creates a stream in order to be passed to the syntax analyzer.
- **CheckSyntaxStructure()** (*attached to the Syntax Analyzer*) checks whether the given input is in the correct syntax based on the programming language's grammar.
- **BuildSyntaxTree()** (*attached to the Syntax Analyzer*) after the check based both on the input and the grammar the syntax tree is constructed and is passed to the semantic analyzer.
- **GenerateSymbolTable()** (*attached to the Syntax Analyzer*) produces the Symbol table.
- **CheckDeclarationsAndStatements()** (*attached to the Semantic Analyzer*) assures that the declarations and statements are semantically correct.
- **CheckTypes()** (*attached to the Semantic Analyzer*) checks if each operator has the suitable type operand.
- **GetAnnotatedSyntaxTree()** (*attached to the Intermediate Code Generator*) receives the annotated AST from Semantic Analyzer.
- **ProduceQuads()** (*attached to the Intermediate Code Generator*) based on the annotated AST, quads of intermediate code are produced.

- **ExportIntermediateRepresentation()** (*attached to the Intermediate Code Generator*) sends the produced quads to Code Optimizer.
- **EliminateDeadCode()** (*attached to the Code Optimizer*) removes the code which does not affect the programs outcome.
- **CompileTimeEvaluation()** (*attached to the Code Optimizer*) applies the constant folding technique.
- **StrengthReduction()** (*attached to the Code Optimizer*) changes the high strength operator with a low strength.
- **ExportOptimisedCode()** (*attached to the Code Optimizer*) exports optimized code to target Code Generator.
- **ProduceMachineCode()** (*attached to the Code Generator*) converts the optimized code into the target code.

6.2.3. Final Result

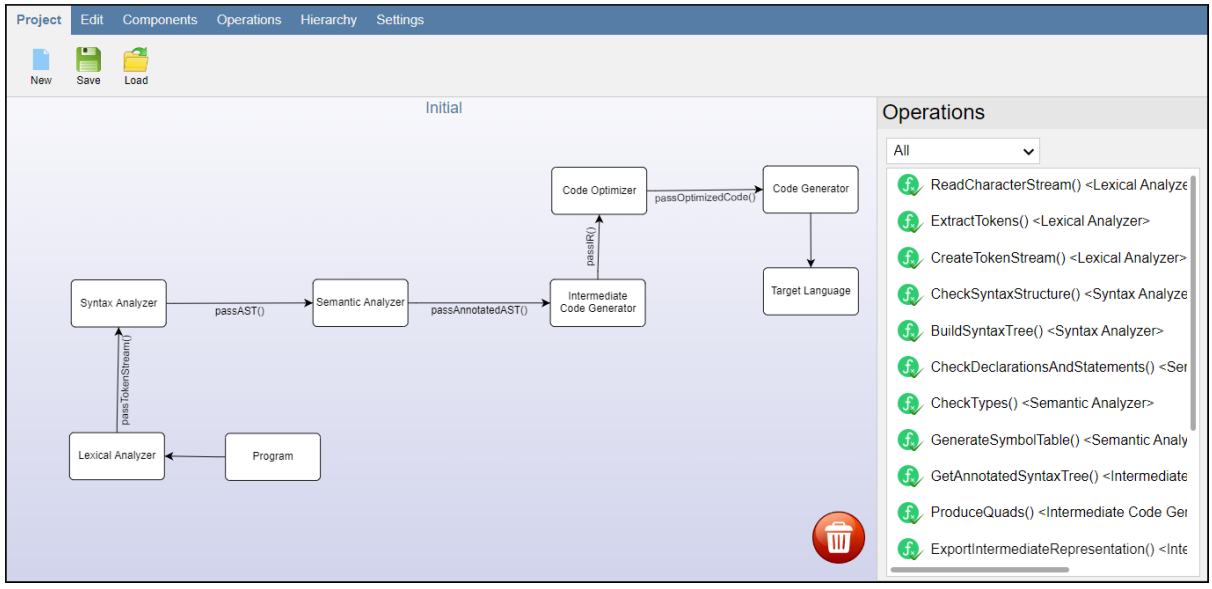

Figure 38: Compiler's Architecture.

6.3. Rapid Software Architecture Designer's Architecture

In this section, our tool's system architecture is being designed and studied. As above, firstly the components are being examined and then the operations. The difference between the other case studies is that now more than one layer of architecture is implemented. More specifically, it is crucial for the Blackboard's and View's role to be studied further.

6.3.1. Components

The components of the architecture have been described on the chapter 3 and 5. However, in order for the case study to be properly presented, they will be again mentioned and described briefly. The components of the initial layer are the following:

- **Observer** is responsible for detecting any input from the user and invoking the corresponding function.
- **Instance Creator** creates, updates and configures the Diagram and its elements.
- **Blackboard** saves and loads the user's actions, the architectural elements' data and the layers of architecture's data. In this component, there are many internal updating and data modifying operations that are invoked from other components.
- **View** constitutes the front end component of the tool.

The Blackboard architectural layer includes:

- **Action Holder** is responsible for storing the user's actions and handling the command, undo and redo stack.
- **Item Holder** is the storage of all the architectural elements' data.
- **Layer Holder** is the storage of all the architectural layers' data.

The View architectural layer includes:

- **Workspace** component represents the area which contains the software architecture's components and links.

- **Toolbar** component is responsible for displaying buttons which correspond to the architecture designing and configuration actions that are described on the chapters 4 and 5. Also, manages the functions that are invoked when these buttons are pressed.
- **Operation's Table** component displays the operations depending whether "All" or "By Component" mode is selected by the user.
- **Hierarchy Tree** handles the insertion and deletion of tree nodes as well as its listeners.

6.3.2. Operations

The system's architecture abstract operations are the following:

- **DetectKeyboardInput()** (*attached to the Observer*) catches the user keyboard buttons' input.
- **DetectMouseInput()** (*attached to the Observer*) catches the user moue buttons' input.
- **InvokeElementSelection()** (*attached to the Observer*) changes the clicked item or items selection state.
- **InvokeContextMenuDisplay()** (*attached to the Observer*) is invoked when a right click takes place and shows the corresponding context menu.
- **InvokeUndoAction()** (*attached to the Observer*) calls undo, when left "Ctrl" key is detected alongside with Z key.
- **InvokeRedoAction()** (*attached to the Observer*) calls redo, when left "Ctrl" key is detected alongside with Y key.
- **InvokeZoomAction()** (*attached to the Observer*) calls the related zoom action depending which direction the mouse wheel is turned when the left "Ctrl" key is pressed.
- **InvokeToggleFullscreen()** (*attached to the Observer*) calls fullscreen mode function.
- **InvokeDeleteWithKey()** (*attached to the Observer*) calls the delete with key function when a delete key input is detected.
- **CreateDiagram()** (*attached to the Instance Creator*) produces the workspace's diagram.
- **CreateLink()** (*attached to the Instance Creator*) produces the graphical representation of a Link (line segment) between two components.

- **CreateComponent()** (*attached to the Instance Creator*) produces the graphical representation of a Component (rounded rectangle).
- **ConfigureElementsListeners()** (*attached to the Instance Creator*) handles the architectural item listeners.
- **EditElementsDiagramName()** (*attached to the Instance Creator*) updates the new element's name given by the user.
- **AlterElementsAppearance()** (*attached to the Instance Creator*) is used to change an item's appearance (color, font size, font family etc.).
- **SendDataToItemHolder()** (*attached to the Instance Creator*) stores elements data to Item Holder.
- **UpdateElementsData()** (*attached to the BlackBoard*) is called from another component in order to update an items' name or description.
- **SendElementsData()** (*attached to the BlackBoard*) sends elements' data to other components.
- **ToString()** (*attached to the BlackBoard*) converts the blackboard component to string.
- **ToObject()** (*attached to the Blackboard*) converts a string to blackboard object.
- **SaveProject()** (*attached to the Blackboard*) saves the project JSON in a ".prj" file.
- **LoadProject()** (*attached to the Blackboard*) loads a project ".prj" file to the tool.
- **InitializeToolbarsButtons()** (*attached to the View*) creates all tab's buttons.
- **InitializeHierarchyTree()** (*attached to View*) creates the layer hierarchy tree.
- **InitializeWorkspace()** (*attached to the View*) creates the tool's workspace.
- **InitializeOperationTable()** (*attached to the View*) creates the operation table.
- **ScrollDiagram()** (*attached to the Workspace*) is called by the panning method in order for the user to move the components and links all at once.
- **HandleBinDeletion()** (*attached to the Workspace*) deletes the selected components that are being dragged on top of the bin.
- **InitializeLayerPath()** (*attached to the Workspace*) creates the layer path on top of the workspace.
- **RefreshPath()** (*attached to the Workspace*) updates the path when the layer is changed.

- **ZoomOutOfDiagram()** (*attached to the Workspace*) is triggered by the observer to zoom out of the diagram.
- **ZoomInDiagram()** (*attached to the Workspace*) as above but the functionality differs.
- **ShowComponentTabButtons()** (*attached to the Toolbar*) displays the action buttons of the component tab, based on component selection.
- **ShowOperationTabButtons()** (*attached to the Toolbar*) displays the action buttons of the operation tab, based on component and operation selection.
- **ShowProjectTabButtons()** (*attached to the Toolbar*) displays the action buttons of the project tab, when it is selected.
- **ShowEditTabButtons()** (*attached to the Toolbar*) displays the action buttons of the edit tab. If a single architectural element is selected then the edit element's data button is displayed, in order for the user to edit the specific element's data.
- **ShowHierarchyTabButtons()** (*attached to the Toolbar*) displays the action buttons of the hierarchy tab. Moreover, the hierarchy tree is displayed and when the user selects one or more components the "Move To" button is shown.
- **ShowSettingsTabButtons()** (*attached to the Toolbar*) displays the action buttons of the settings tab.
- **AddClickListenersToButtons()** (*attached to the Toolbar*) adds click listeners to all the buttons.
- **InvokeComponentTabAction()** (*attached to the Toolbar*) invokes the corresponding method when a component's tab button is clicked.
- **InvokeOperationTabAction()** (*attached to the Toolbar*) invokes the corresponding method when an operation's tab button is clicked.
- **InvokeProjectTabAction()** (*attached to the Toolbar*) invokes the corresponding method when a project's tab button is clicked.
- **InvokeEditTabAction()** (*attached to the Toolbar*) invokes the corresponding method when an edit's tab button is clicked.
- **InvokeSettingsTabAction()** (*attached to the Toolbar*) invokes the corresponding method when a setting's tab button is clicked.

- **InvokeHierarchyTabAction()** (*attached to the Toolbar*) invokes the corresponding method when a hierarchy's tab button is clicked.
- **ShowAllOperations()** (*attached to the Operations' Table*) is invoked when "All" mode is selected.
- **ShowCurrentOperations()** (*attached to the Operations' Table*) is invoked when "By Component" mode is selected.
- **AdjustRightTabWidth()** (*attached to the Operations' Table*) is invoked when the user drags the edge of the operation tab in order to adjust its width.
- **InsertOperationToTheTable()** (*attached to the Operations' Table*) is called to add an operation to the operation list.
- **RemoveOperationFromTheTable()** (*attached to the Operations' Table*) is called to remove one or more operations from the operation list.
- **RefreshTree()** (*attached to the Hierarchy Tree*) updates the hierarchy tree when a change is made.
- **InsertNodeToTree()** (*attached to the Hierarchy Tree*) adds a node to the tree when a new layer is created.
- **RemoveNodeFromTree()** (*attached to the Hierarchy Tree*) removes a node from the tree when a layer is deleted.
- **AddNodeListener()** (*attached to the Hierarchy Tree*) inserts a click listener to the tree nodes for layer change.
- **ToggleTreeAppearance()** (*attached to the Hierarchy Tree*) toggles hierarchy tree appearance.
- **CollapseNode()** (*attached to the Hierarchy Tree*) collapses tree path.
- **ExpandNode()** (*attached to the Hierarchy Tree*) expands tree path.
- **AdjustLayerTabWidth()** (*attached to the Hierarchy Tree*) is invoked when the user drags the edge of the layer tab in order to expand it and adjust tab's width.
- **AddItem()** (*attached to the Item Holder*) inserts an architectural element's item object in the item holder.

- **DeleteItem()** (*attached to the Item Holder*) deletes an architectural element's item object in the item holder.
- **LinkComponents()** (*attached to the Item Holder*) inserts link object to item holder and updates the components data.
- **UnlinkComponents()** (*attached to the Item Holder*) deletes link object from item holder and updates the components data.
- **DetatchOperationFromComponent()** (*attached to the Item Holder*) updates the components data, by deleting the operation's id from the component's function array.
- **UpdateItemsDetails()** (*attached to the Item Holder*) is invoked in order to internally update elements' name and description.
- **ProduceHierarchyTreeJSON()** (*attached to the Layer Holder*) creates the JSON that represents the node in the hierarchy tree.
- **AddLayer()** (*attached to the Layer Holder*) inserts a layer in the Layer Holder.
- **SelectLayer()** (*attached to the Layer Holder*) selects an architectural layer, which its workspace and operation table will replace the current ones.
- **DeleteLayer()** (*attached to the Layer Holder*) removes the layer from the layer holder.
- **SaveItemHolder()** (*attached to the Layer Holder*) save the changes concerning the item holder into the related layer.
- **GetItemHolder()** (*attached to the Layer Holder*) returns layer's Item Holder.
- **SaveCommand()** (*attached to the Layer Holder*) saves the latest user action in the command and in the undo stack.
- **EmptyRedoStack()** (*attached to the Layer Holder*) clears redo stack.
- **Undo()** (*attached to the Layer Holder*) undoes user's latest action by calling the inverse action.
- **Redo()** (*attached to the Layer Holder*) redoes user's latest undone action.

6.3.3. Final Result

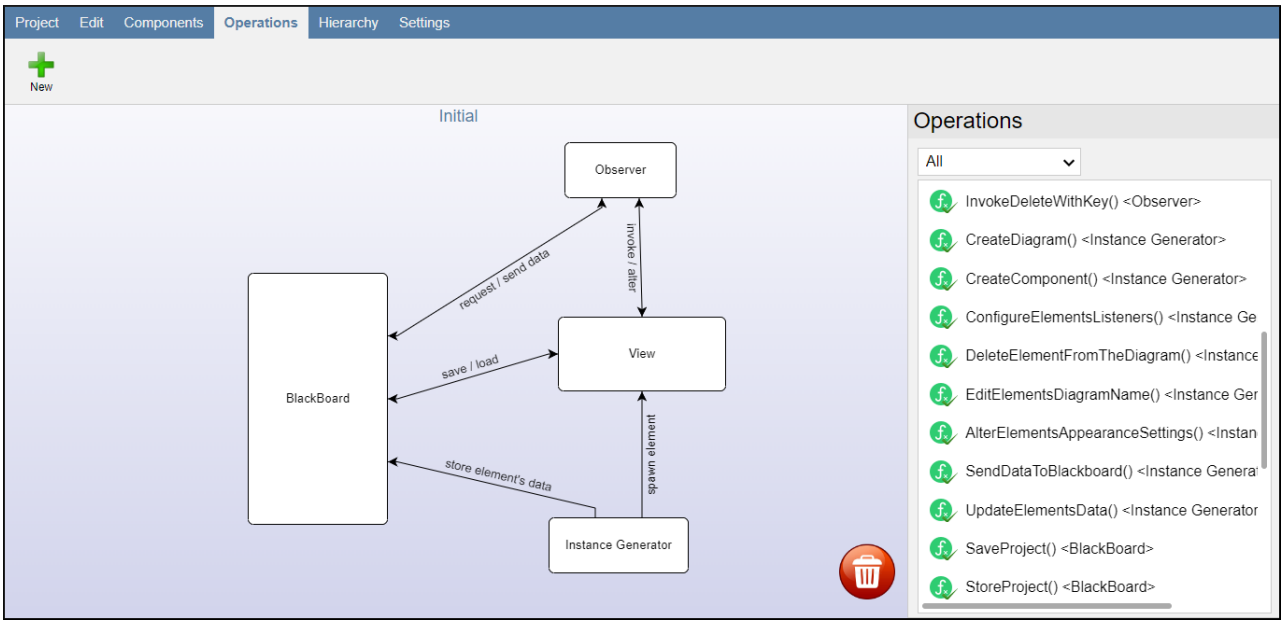

Figure 39: System Architecture's Initial Layer.

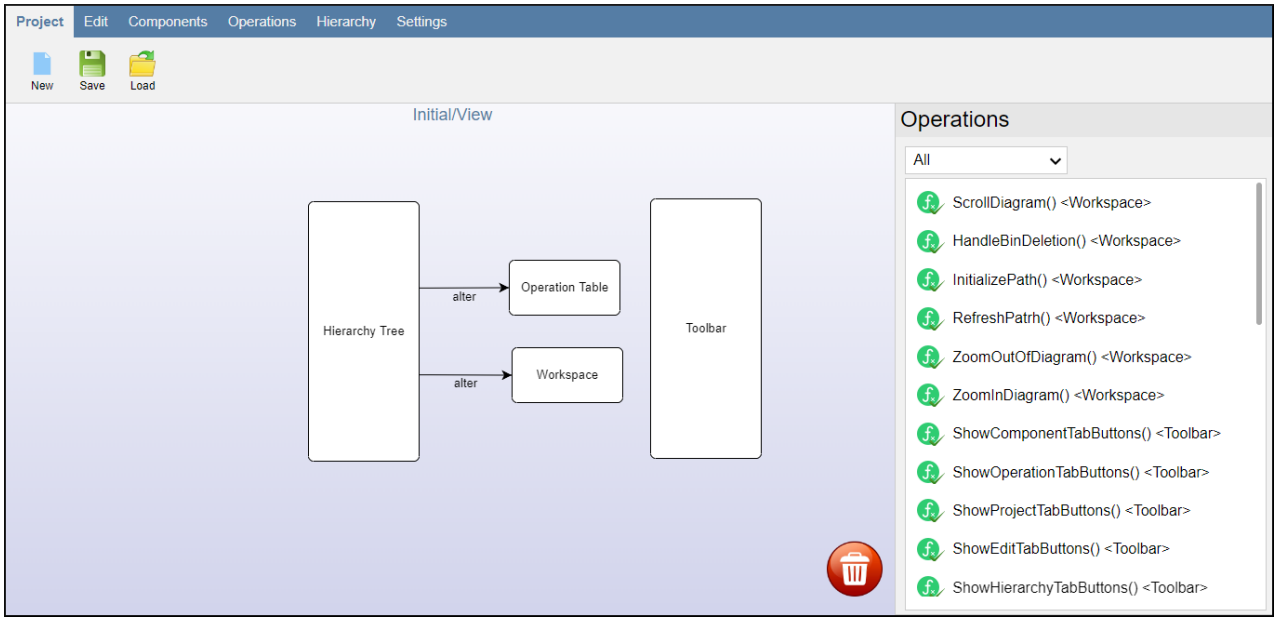

Figure 40: System Architecture's View Layer.

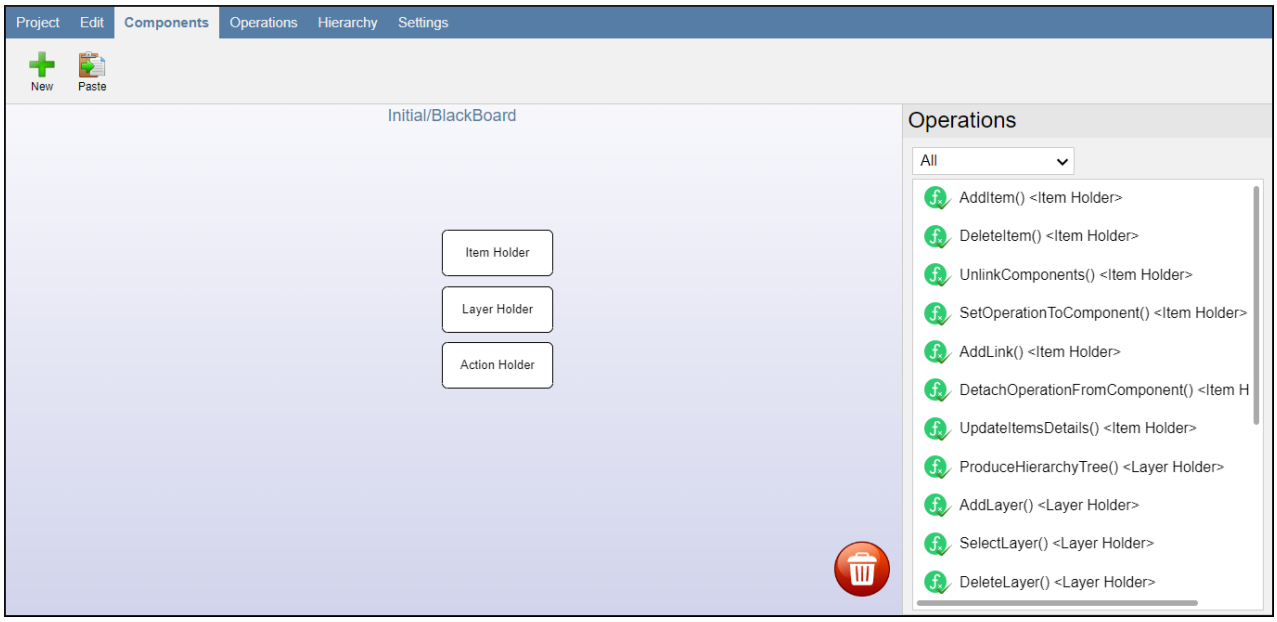

Figure 41: System Architecture's Blackboard Layer.

Chapter 7

7. Conclusion and Future Work

It is mandatory for systems to be based on a correctly built software architecture because not only it acts as a link between the requirements and the implementation, but also it offers a wide variety of benefits. The process of software architecture designing is defined by its simplicity and abstraction. When more information and details than necessary are added, these two properties are distorted, maximizing the danger of not having the desired outcome. Furthermore, the user spends a considerable amount of time choosing the best graphical representation for the architectural elements and filling excessive data fields for their creation.

Our work's tool provides a solution to this matter by reducing significantly these data fields, requiring only a name and an optional description for an architectural element to be created. Moreover, we offer only one graphical representation of each architectural element. This way, the user can only focus on making correct architectural decisions and building faster software architectures, without worrying about details and information, that are not important at this phase of software development.

Apart from the creation of architectural elements, our tool offers the user the ability to manipulate them. More specifically, components can be deleted, joined, linked, or split based on their operational criteria. Some components can be subdivided for their role to be examined further. On the other hand, the operations, which represent the operational features of the designing system, can be attached to a component or unattached from it, depending on its relation with the component's role.

For testing and validation purposes, we have applied three different case studies, which examine three different features of the tool. Firstly, a game engine architecture has been created, where the main component calls operations from the other components while executing the main game loop. The second case study concerns a compiler's architecture. In this case study, the sequential pattern is being applied, starting from the user's source code and after processing, ending with the target code. Finally, our tool's system is being constructed and consists of three layers of architecture: a) the Initial, b) the View and c) the Blackboard layer.

Conclusively, would be ideal for our work's tool, to be compatible with touch screen monitors, tablets, and smart boards, as they are being widely used by both the industrial and academic community. If there are any design patterns it would be very helpful to be recognized and highlighted from the layer info modal. Instead of only detecting the design patterns, our tool can be expanded to detect bad practices and offer suggestions to eliminate them. Finally, it would be very convenient if our tool were connected with a cloud service, so that the users may more easily share and exchange our tool's project and configuration files.

Bibliography

[1] Garlan, D. (2003). Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In Lecture Notes in Computer Science (pp. 1–24). Springer Berlin Heidelberg.

[2] Garlan, D. (2000). Software architecture. In Proceedings of the conference on The future of Software engineering.

[3] Savidis, A., & Peris, A. (2022). Rapid Interactive Software-Architecture Design with Split-n-Join Actions. In Human Interaction and Emerging Technologies (IHIET 2022): Artificial Intelligence and Future Applications. 8th International Conference on Human Interaction and Emerging Technologies. AHFE International.

[4] Bosch, J. (2004). Software Architecture: The Next Step. Lecture Notes in Computer Science, 194–199.

[5] Soni, D., Nord, R. L., & Hofmeister, C. (1995). Software architecture in industrial applications. In Proceedings of the 17th international conference on Software engineering- ICSE ’95. the 17th international conference. ACM Press.

[6] Star UML – The Open Source UML/MDA Platform. Version 5.0.2. Released on June 14, 2022. Developer Team: MKLabs Co. Ltd. Official Website: https://staruml.io/ Accessed Online: 07/2022.

[7] Astah – The power of modeling software. Version 8.5. Released on March 10, 2022. Developer Team: Change Vision. Official Website: https://astah.net/ Accessed Online: 07/2022.

[8] Altova UModel 2022 – UML tool for software modeling and application development. Version R2. Released March 8, 2022. Development Team: Altova. Official website: http://www.altova.com/umodel.html. Accessed Online: 07/2022.

[9] Enterprise Architect Enterprise Architect – Visual Modeling Platform. Version 15.2. Released on 08/2000. Development Team: Sparx Systems. Official website: http://www.sparxsystems.com/products/ea/index.html. Accessed online: 07/2022.

[10] Archi — The Open Source modeling toolkit for creating ArchiMate models and sketches. Used by Enterprise Architects everywhere. Version 4.9.1. Released on October 26, 2021. Developer Team: Phil Beauvoir, Jean-Baptiste Sarrodie. Official website: https://www.archimatetool.com/. Accessed online: 07/2022.

[11] Gaphor — Modeling for Everyone. Version 2.0.1. Released on August 27, 2020. Developer Team: Arjan Molenaar & Dan Yeaw. Official website: https://gaphor.org/. Accessed online: 08/2022.

[12] Kazman, R. (1996). Tool support for architecture analysis and design. Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints ’96) on SIGSOFT ’96 Workshops.

[13] Systems Modeling Language SysML by OMG. Official website: http://www.omgsysml.org/. Accessed online: 08/2022.

[14] Business Process Model and Notation BPMN by OMG. Official website: http://www.bpmn.org/. Accessed online: 08/2022.

[15] The Structurizr DSL by Structurizr. Official site: https://github.com/structurizr/dsl/blob/master/docs/language-reference.md. Accessed online: 08/2022.

[16] Structurizr. Version: 1.19.1. Released on 30 March, 2022. Development Team: Structurizr Limited. Official website: https://structurizr.com/. Accessed online 08/2022.

[17] Mermaid by Knut Sveidqvist. Official website: https://mermaid-js.github.io/mermaid/#/. Accessed online 09/2022.

[18] DOT Language by The Graphviz. Official website: https://graphviz.org/doc/info/lang.html. Accessed online: 09/2022.

[19] Plant UML by Arnaud Roques. Official website: https://plantuml.com/. Accessed online: 09/2022.

[20] Plant UML C4 library by Arnaud Roques. Official website: https://plantuml.com/stdlib/. Accessed online: 09/2022.

[21] Diagrams.net — Security-first diagramming for teams. Released on September 1, 2022. Version: 20.2.8. Developed by JGraph Ltd. Official website: https://www.diagrams.net/. Accessed online: 09/2022.

[22] Lucidchart — Where seeing becomes doing. Released on December 2008. Developed by Lucid Software Inc. Official website: https://www.lucidchart.com/. Accessed online 08/2022.

[23] GoJS – Interactive JavaScript Diagrams for the Web. Version 2.2. Released on January 20, 2022. Development Team: Northwoods Software. Official Website: https://gojs.net/latest/index.html Accessed online: 09/2022.

[24] Wikipedia, “Blackboard design pattern”. Available: https://en.wikipedia.org/wiki/Blackboard_(design_pattern). Accessed online: 09/2021.

[25] Wikipedia, “JSON format – JavaScript Object Notation”. Available: https://en.wikipedia.org/wiki/JSON. Accessed online: 09/2021.

[26] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. In Communications of the ACM (Vol. 15, Issue 12, pp. 1053–1058). Association for Computing Machinery (ACM). Accessed online: 05/2022.

[27] Garlan, David., & Shaw, Mary. (1994). An Introduction to Software Architecture (CMU/SEI-94-TR-021). Retrieved September 26, 2022, from the Software Engineering Institute, Carnegie Mellon University website: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12235. Accessed online: 11/2021.

[28] JSTree – jQuery Tree Plugin. Version 3.3.12. Released September 3, 2021. Development Team: Ivan Bozhanov. Official Website: https://www.jstree.com/. Accessed Online 04/2022.

[29] Wikipedia, “Tooltip” – Available: https://en.wikipedia.org/wiki/Tooltip. Accessed Online 05/2022.

[30] Eclipse IDE—An integrated development environment (IDE) used in computer programming. Released on 11/2001. Development Team Eclipse Foundation. Official

Website https://www.eclipse.org/ide. Accessed online: 08/2022.

[31] Visual Studio—It's how you make software. Version 17.3. Released on August 9, 2022. Development Team: Microsoft. Official Website: https://visualstudio.microsoft.com/. Accessed online 08/2022.

[32] A. Savidis, "Games Basic Architecture." [Online]. Official Website: http://www.csd.uoc.gr/~hy454. Accessed online 08/2022.