Refactoring a Complex GUI Application:

A Case Study with the Auckland Layout Editor

Lingjun (Irene) Zhang

Supervised by:

Christof Lutteroth

A report for Compsci 380 course, The University of Auckland, 2014.

Abstract

Refactoring is a collection of actions that aims to improve software quality without breaking or changing existing functionality. It is generally considered to improve quality attributes such as understandability and maintainability. However there is a lack of controlled studies to assess the effect of refactoring on graphical user interface (GUI) applications. Scala is a general-purpose, multi-paradigm programming language, which combines functional and imperative programming styles. It is claimed to be a superior language compared to Java for the development of GUI applications. There is an absence of investigations comparing Java and Scala Swing.

In this project, we address these gaps and investigate the effects of refactoring and the use of Scala in a GUI application, using the Auckland Layout Editor (ALE) as a case study. ALE is a GUI builder that uses the Auckland Layout Model (ALM), a constraint-based layout manager. We converted the source code from Java into Scala and performed some refactoring to achieve more separation of concerns (SoC). We evaluated the effect of refactoring using internal metrics. Comparing the metrics at the start of the project and at the end of the project seems that software quality was improved in the areas of maintainability, reusability, understandability, and extensibility. We also compared key Java and Scala constructs to assess which ones are better suited for programming a complex GUI application such as ALE. We found Scala to provide a better capability to decompose a complex application into simpler parts. Other features of Scala that we felt were beneficial include: its concise syntax as compared to Java, a more powerful event handling system, a more flexible inheritance structure and its support for implicit collection transformations, which are not present in Java.

In addition to conversion to Scala and refactoring of ALE, the following contributions were made: ALE's existing functionality was improved and new functionality was implemented, and a prototype plugin for the IntelliJ IDEA was created.

Table of Contents

List of Figures

List of Tables

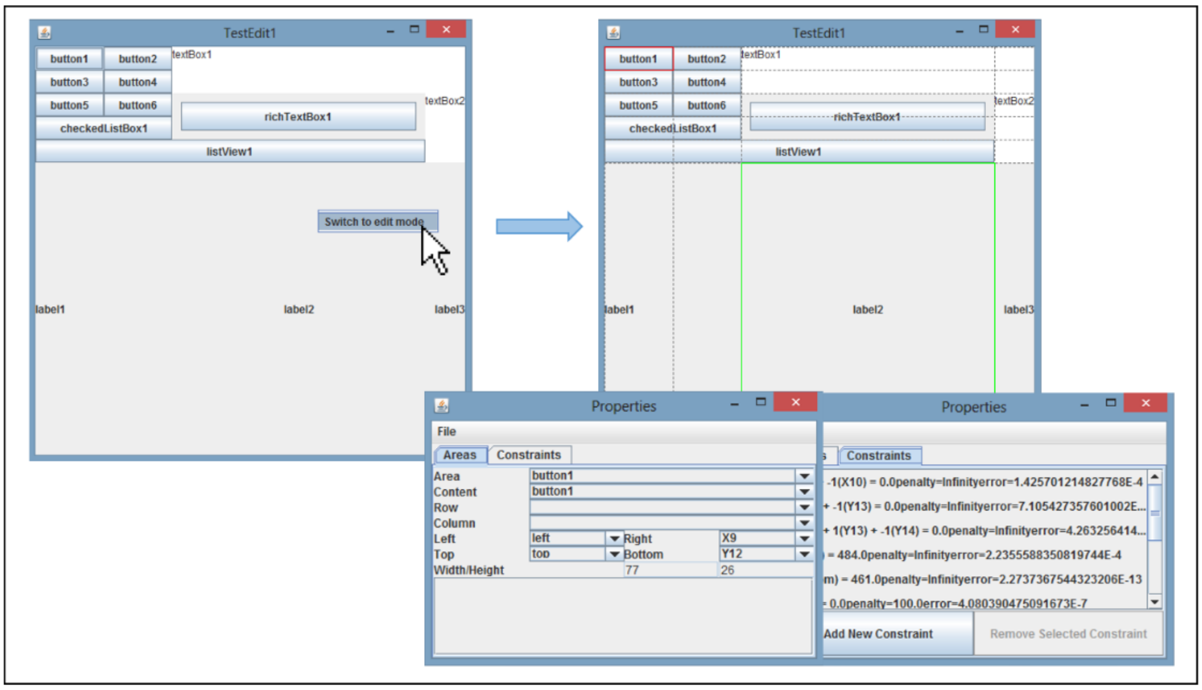
Chapter 1. Introduction

Graphical user interfaces (GUIs) have become increasingly important as the number of applications using them increased over the last decades. Almost all applications are built by using WYSIWYG (What You See Is What You Get) builders, which are supported by GUI toolkits. These tools can be used to add various components to a canvas and perform various operations on them (such as resize and move). Layout engines are often incorporated into modern GUI toolkits to specify the dynamic behaviour of a layout. This is achieved by a layout manager implementing a layout model, which takes user specifications to set the optimal position and size of components in the layout.

The Auckland Layout Editor is similar to other GUI builders in that components can be dragged and dropped from a palette to a canvas. ALE uses the Auckland Layout Model [1], a layout engine that uses constraint-based layouts. It is a powerful and flexible layout model that uses constraints to define the positions of components. However, constraint-based layouts are complex: editing individual constraints requires specialist knowledge and therefore errors such as overlapping and the presence of unnecessary constraints are likely to occur. ALE simplifies editing for the end-user by hiding the lower-level details such as specifying constraints. Instead it allows users to specify constraint-based layouts using only simple mouse operations [2] such as moving, swapping, inserting and deleting, while maintaining all specifications and non-overlap between components in the layout.

Besides the original C++ version, a prototype Java version of ALE using the constraints-based layout system as described above has also been developed (for a full description see Chapter 5). Like the original ALE, it is also a drag and drop type editor which supports the insertion of a particular component into a testing window canvas. It uses the ALM layout manager as in the original ALE (see Chapter 4). Every GUI is started in the operational mode (normal function); a user can switch a GUI into editing mode at any time during run-time, which allows for its customization [3, 4]. Figure 1-1 shows the switch to editing mode by clicking the “Switch to Edit Mode” menu item in the GUI testing window named “TestEdit1” (see Section 5.3 for more details on the switch process), accompanied by the appearance of a properties GUI editor window (the properties window).

The properties window can be in one of two modes: area mode, which allows a user to edit the rectangular areas containing the components of the GUI, and constraints mode, which allows a user to edit the constraints in the layout specification. The GUI window (also referred to as the testing window) and the properties window are synchronized so that each window is automatically updated to reflect the settings in the other one when a change is made. It supports some of the edit operations seen in the original ALE version such as inserting, removing, swapping and resizing (see Section 5.5). Some other features are not yet supported.



Area	button1		
Content	button1		
Row			
Column			
Left	left	Right	X9
Top	top	Bottom	Y12
Width/Height		77	26

```
-1(X10) = 0.0penalty=Infinityerror=1.425701214827768E-4
+ -1(Y13) = 0.0penalty=Infinityerror=7.105427357601002E...
+ 1(Y13) + -1(Y14) = 0.0penalty=Infinityerror=4.263256414...
) = 484.0penalty=Infinityerror=2.2355588350819744E-4
m) = 461.0penalty=Infinityerror=2.2737367544323206E-13
= 0.0penalty=100.0error=4.080390475091673E-7
```

Figure 1-1 Switching a GUI from operational mode into editing mode

In this paper, we use the ALE Java version as a case study to address the following research questions:

1) Does converting a complex GUI application (such as ALE) from Java to Scala improve its code base?
2) How can a complex GUI application (such as ALE) be refactored?
3) In how far does refactoring help to improve the quality of a complex GUI application (such as ALE)?

Java is a widely-used general-purpose object-oriented (OO) language. Despite its popularity, it has some limitations as a language, which are addressed by the relatively new Scala language. Scala is a general-purpose programming language with multiple paradigms, supporting both object-oriented and functional programming approaches (see Chapter 3 for an introduction to the language). Scala Swing is a wrapper system around Java Swing and is considered to be more powerful due to its cleaner syntax without using explicit inheritance [5]. Its reactions system, which uses partial functions, also has several advantages compared to Java's event handling system. To take advantage of the various benefits of Scala Swing, we converted the source code of the Java version of ALE to the Scala language. At the time of this report's writing, there were no studies directly comparing Scala and Java GUI applications, therefore the differences of a complex GUI application using Java Swing versus using Scala Swing were investigated in this case study.

Software quality is usually defined as the embodiment of particular combination attributes such as maintainability, reusability and comprehensibility [6]. However, these so-called external/indirect quality attributes tend to be subjective in nature and difficult to measure. Therefore much research effort has focused on indirectly assessing the degree to which a software product possesses these attributes, by using a set of measurable internal/direct quality attributes [7-12]. These include cohesion, coupling, complexity, and inheritance [13].

Refactoring is defined as a transformation of an object-oriented program by redistributing classes, variables and methods in the class hierarchy that improves the design without changing its behaviour of functionality [14]. In this project, the edit operations of ALE were refactored to achieve a better separation of concerns (SoC). Several studies have evaluated whether refactoring is beneficial or detrimental to software product quality characteristics. These studies were conducted by either assessing internal quality attributes directly [15, 16] or by using statistics [17-19] or internal attributes [8, 9, 20-24] as predictors of external quality attributes (as outlined in Section 2.2). To extend upon these findings, the ALE Java version was used to evaluate the effect of refactoring on software quality, measured by both direct quality attributes and other related statistics.

Furthermore, some edit operations were added to extend the functionality of ALE Java version and others modified to improve usability. One future goal is to make the stand-alone ALE a plugin for an Integrated Development Environment (IDE). Therefore a prototype plugin of ALE was developed in this project for the IntelliJ IDEA, which was chosen due to its support for Swing and the Scala language. In conclusion, the four major goals of this project are as follows:

1) To convert the source code of ALE Java version to Scala
2) To refactor the source code to improve its quality
3) To implement new functionality and improve the usability of some existing functionalities
4) To develop a prototype plugin for the IntelliJ IDEA
5) To evaluate the effects of the aforementioned steps

It must be noted that even though some regard the conversion of code into another language to be a subtype of refactoring, the conversion of the source code and refactoring efforts are described and evaluated separately in this report.

The remainder of this paper is structured as follows: Chapter 2 outlines the related work in terms of studies involving refactoring and comparing Scala and Java. Chapter 3 offers an introduction to the Scala language. Chapter 4 describes ALM and its important classes. Chapter 5 provides an overview of the ALE prototype, including its layout system, class structure, features and GUI. Chapter 6 describes how ALE was converted to Scala, how it was refactored and modified during the course of the project, and outlines the steps taken to create the IntelliJ plugin. Chapter 7 critically examines the final product, comparing quantitative and qualitative measures of quality before and after the project. In addition, our experiences with the Scala language during the course of this project is described. Chapter 8 summarizes future directions and Chapter 9 concludes this paper.

Chapter 2. Related Work

In this chapter previous work relating to the objectives of this study is explored. First, an overview of software quality is given with definitions of internal and external quality attributes that are relevant for this study. In addition, studies investigating the measurement of the former as a predictor of the latter are summarized. Next, existing studies about the effect of refactoring on software quality are explored. Finally, existing research comparing Scala and Java are reviewed.

2.1. Software Quality

Object-oriented metrics to quantitatively measure internal quality attributes have been researched and classified by many authors [25-28]. They are all potentially relevant to this study since the majority of ALE is written in the object-oriented style. Since the number of possible metrics described in the literature is virtually endless, only the most popular ones are considered. They are summarized in Table 1 along with the internal quality attributes they measure. They are primarily from the Chidamber & Kemerer (CK) metrics suite [27] but also from the Traditional set [29] and MOOD set [30].

Internal quality attribute	**Description**	**Metrics**
Cohesion	Measures the level of dependency of the local methods of a class on each other and the degree to which they are related and work together to provide well-bounded behaviour [31].	Lack of Cohesion of Methods (LCOM): “the number of disjoint/non-intersection sets of local methods” [27]
Coupling	A measure of the degree of interdependency between modules/entities [28]. Two classes are coupled when one class uses another’s methods and/or instance variables.	Coupling Between Object classes(CBO): Number of classes to which a given class is coupled [27]
		Response for a Class(RFC): “number of methods a that can potentially be executed in response to a message received by an object of that class.”[27]
Modularity	A module has its own area of responsibility (high cohesion) and communication between those parts is scarce and happens through well-defined interfaces (loose coupling) [32].	Uses Coupling and Cohesion metrics
Complexity	Defined as the psychological complexity associated with a software and is a predictor of the time and effort required to maintain it [27].	Cyclomatic Complexity(CC): “the complexity of a control flow graph of a method” [29]
		Weighted Methods per Class(WMC): This metric is the sum of complexities of methods defined in a class where all “method complexities are considered unity” [27]
Encapsulation	Describes a type of design in which interaction with an object can only be achieved through its public interface [33]. It is a mechanism to achieve data abstraction and information hiding.	Method Hiding Factor (MHF) or Attribute Hiding Factor (AHF): How well-hidden methods or attributes are within a particular class, respectively[30]

Inheritance	Inheritance is a mechanism in which a class acquires characteristics from another [34].	Depth of Inheritance Tree(DIT): Maximum inheritance path from class to root class [27]
		Number of Children (NOC): Number of direct descendants for each class [27]
Others	Lines of Code (LOC), Source Lines of Code (SLOC) Comment Percentage (CP) – all from the Traditional suite	

Table 1 Summary of the definitions of internal attributes used in this study, with their associated metrics

Software quality is defined as the degree up to which a system possesses certain external attributes or characteristics. Their taxonomy was initially introduced by ISO9126 [35] which present these characteristics: functionality, reliability, usability, efficiency, maintainability and portability. Other characteristics have also been described in literature such as: performance, usability, portability, understandability, reusability, and readability [13]. Much research has gone into determining the degree to which certain internal quality attributes to predict external quality attributes, and so allow their measurement indirectly. Du Bois et al. developed practical guidelines for improving cohesion and coupling metrics and validated them as indicators for maintainability on an open source software system [7]. Kataoka et al. [8] proposed a quantitative evaluation method using coupling metrics to measure changes to maintainability through refactoring. A method for assessing refactoring effect on reusability was proposed by Moser et al using a set of internal software metrics such as CBO, LOC and CC[9]. Testability was linked to a set of internal quality metrics by Bruntink and Deursen [10]. Dandashi [11] demonstrated that adaptability, maintainability, comprehensibility and reusability may be deduced from the analysis of internal metrics such as WMC, CBO, RFC and CC. The relationship between internal metrics and fault-proneness has also been studied [12]. These findings are summarized in Table 2; note that many more other studies have correlated internal metrics with software quality or validated the significance of other such studies in literature; these studies are emphasised because they have been associated with refactoring in literature.

Study	**Measured internal metric(s)/attribute**	**Coupled external attribute**
Bois et al. [7]	Cohesion and coupling	Maintainability
Kataoka et al [8]	Coupling	Maintainability
Moser et al. [9]	CC and CK metric suite	Reusability
Bruntink and Deursen [10]	Lines of code per class (LOCC), NOC, number of fields (NOF), number of methods (NOM), RFC, WMC	Testability
Dandashi [11]	CC, number of Physical Source Statements (PSS), WMC, DIT, NOC, RFC, CBO	Adaptability, Maintainability, Comprehensibility, Reusability
Gyimothy et al. [12]	Many metrics including WMC, DIT, RFC, NOC, CBO, LCOM , LOC	Fault-proness
Basili et al. [36]	NOC, CBO, RFC	Fault probability

Table 2 Summarizing studies which investigate the use of internal software attributes as predictors of external software attributes

2.2. Existing Studies of Refactoring

In this section, empirical studies investigating the effect of refactoring on internal and external software attributes are explored.

Some researchers assessed refactoring effects on the internal software quality attributes through measuring internal metrics only. Stroggylos and Spinellis analyzed how documented refactorings in four popular open source software systems affected their code metrics and concluded that refactoring sometimes has detrimental effects if not used effectively [15]. Du Bios and Mens [16] evaluated the effect of three types of refactoring (extract method, encapsulate field, and pull-up method) on internal software measures such as NOM, NOC, coupling, and cohesion and found both positive and negative impacts on the measures.

Others assessed refactoring effects on external software quality attributes by surveying programmer experience or measuring certain statistics within the scope of their study. By performing a controlled experiment on the differences in program comprehension between using the Refactor to Understand and Read to Understand patterns, Du Bois et al. [17] provided the first empirical support for the claim that refactoring increases comprehensibility of code. Changeability was the topic of another study by Geppert et al. which showed that change effort (decrease by 11%) and customer reported defects were significantly reduced after refactoring [18]. An empirical evaluation on maintainability (measured by the time required to fix random errors in code) and modifiability (measured by time and LOC needed to implemented new requirements) was conducted by Wilking et al. [19] without improvement to both.

By far the most common approach involved assessing refactoring impact on internal attributes as indicators of changes in external software attributes (drawing on the conclusions of studies such as those in Table 2). Using a quantitative evaluation method, Kataoka et al. found that refactoring enhances system maintainability [8]. Leitch and Stroulia also studied maintainability by evaluating two types of refactoring (extract method and move method) on two software systems produced reduction in code size, density of dependencies and regression testing. The conclusion was that refactoring improves maintainability [20].

In an academic study by Stroulia and Kapoor, a case study performed on a group of students demonstrated that size and coupling metrics decreased after refactoring, improving the extensibility of the software [21]. Moser et al. conducted an empirical study on a commercial software system with the CK metrics suite and the CC metric from the Traditional suite used as indicators for reusability. It was found that refactoring enhanced reusability of hard-to-reuse classes in an XP-like development environment [9]. A later study by Moser et al. measured LOC, CK metrics, in association with effort (in hours) with conclusion that refactoring reduces complexity while increasing productivity and cohesion [22].

Tahvildari et al. used a software re-engineering quality framework on four case studies and found improvement in maintainability. They also investigated the use of metrics to detect design flaws [23]. Recently, a more comprehensive study was conducted involving several external quality attributes (adaptability, maintainability, comprehensibility, reusability, and testability). No consistent trends were found regarding the effect of refactoring on these quality attributes thus no firm conclusions could be made about whether refactoring improves software quality [24]. Table 3 summarizes the studies mentioned above, and indicate that the majority of studies found that refactoring produces positive effects on code quality.

Studies measuring internal/direct attributes directly			
Study	**Case study**	**Internal quality attributes**	**Result after refactoring**
Stroggylos and Spinellis [15]	Open source software systems	CK metrics, Afferent coupling, Number of Public Methods of a Class (NPM)	Advantageous or disadvantageous depending on the refactoring
Du Bois and Mens [16]	A small demo program	NOM,NOC,CBO,RFC,LCOM	Advantageous or disadvantageous depending on the attribute
Studies measuring external/direct attributes by using statistics within the scope of their study			
Study	**Case study**	**External quality attributes**	**Result after refactoring**
Bois et al. [17]	Controlled study involving two groups of students with similar skills	Comprehensibility	Improved

Geppert et al. [18]	A project in industrial environment	Changeability	Improved
Wilking et al. [19]	Controlled experiment with students	Maintainability and Modifiability	No benefit to both
Studies measuring internal/indirect attributes as indication of external/direct attributes			

Study	**Case study**	**Internal attributes/metrics**	**External attribute**	**Result after refactoring**
Kataoka et al. [8]	A C++ program	Coupling	Maintainability	Improved
Leitch and Stroulia [20]	Two Java case studies	Code size, number of procedures	Maintainability (efforts and costs)	Improved
Stroulia and Kapoor [21]	A research prototype tool	Size and coupling	Extensibility	Improved
Moser et al. [9]	A project in industrial environment	CC, CK metrics, and LOC	Reusability	Improved
Moser et al. [22]	A project in industrial environment	CK metrics, Effort (hour), and LOC	Productivity	Improved
Tahvildari et al. [23]	Four open-source software systems	Complexity, coupling, complexity, and inheritance metrics	Maintainability	Improved
Alshayeb et al. [24]	Three small open-source projects	CK metrics, FOUT, NOM, LOC	Adaptability, Maintainability, Comprehensibility, Reusability, Testability	Inconclusive

Table 3 Summary of studies investigating the effect of refactoring on software quality

2.3. Studies of Scala versus Java

There have been relatively few controlled studies to evaluate the benefits of Scala vs. Java quantitatively and empirically and analyse Scala and Java together. In [37], memory behaviour of programs written in Scala and Java were compared by running the DaCapo [38] benchmark suite. A number of features were analysed using Java Virtual Machine (JVM) profilers including garbage collector workload, object churn, and how the size of the object and immutability influences performance. Another study [39] adapted a benchmark for Scala and Java and ran them on a mobile device and compared the memory usage, power consumption, execution time, and size of application. These studies compare Scala 'sand Java's performance, which are not very relevant to the purpose of this paper and will not be discussed further.

Scala's developers claims that Scala can reduce LOC to about 50% of the same program written in Java, and in extreme cases, Scala's LOC is ten times smaller compared to Java [5] . However, these claims are over-generalized without controlled studies to support them. To date, investigations into claims about the advantages of using Scala instead of Java has only been investigated by one paper [40] and were briefly considered in another paper [41]. To our knowledge, there are currently no studies directly addressing the benefits of Scala for code refactoring and especially in the context of writing GUI applications.

One recent study [40] compared the experiences of thirteen computer science masters students and one industry software engineer who worked on three projects in Scala and Java. The resulting 39 Scala programs and 39 Java programs involved parallel programming and made use of the concurrency library. Key features such as effort, code, language, usage, and performance and programmer satisfaction were analysed. They proved with statistical significance that Scala code is slightly more compact than Java code. Scala had median of 533 lines of code and Java had median of 547, only a median and mean difference of 2.6% and 15.2% for the LOC, respectively.

Scala is also claimed to reduced the time required to debug and test applications compared to Java [5]; however, this claim was challenged in the findings. The tested subjects spent more time developing the Scala code because the automated type inference produced errors while debugging that required effort to track, understand and correct the type. Furthermore, the subjects also required more time to produce a working Scala program compared to Java program due to the increased effort required to understand how to make use of the functional style in the program. The findings indicated that more time was required to solve the problem in Scala compared to Java, the median hours spent on Scala and Java were 56 hours and 43 hours respectively [40].

Another study compared the characteristics of multiple languages (C++, Java, Go and Scala) such as code complexity, the compilation time, run-times, and memory footprint. The same well-defined algorithm was implemented in all four languages, involving several data structures, memory allocation schemes and iteration without language-specific adaptations or optimizations. The findings showed that Scala has the most concise notation and the best potential for optimization of code complexity but suffered from complicated and unpredictable garbage-collection system due to its use of JVM [41].

Chapter 3. Introduction to the Scala language

An important objective of this project is to convert the source code of ALE from Java to the Scala language. Therefore, this chapter aims to give a brief overview of Scala, namely its multi-paradigm nature, its benefits compared to Java, and its GUI toolkit.

3.1. Overview

Scala unifies the object-oriented programing (OOP) paradigm with the functional programming (FP) paradigm. [5]. Created by Martin Odersky and his research group at EPFL, the first public version was released in January 2004; at the time of this report's writing, the latest stable release is 2.10.3. It is a statically typed, mixed-paradigm, JVM language (generates JVM byte code) with succinct and flexible syntax. It is a relatively new language that was created as an alternative to other common statically typed languages like Java and C# to improve upon their lack of scalability and support for component abstraction and composition [42]. Many mainstream companies are migrating to Scala and achieving a boost in productivity, for example LinkedIn, Novell, Xerox, Sony Pictures Imageworks [43] and Twitter [44].

3.2. Scala's Multi-Paradigm Programming

Throughout the history of software development there has always been a need for language that are able to deal with growing complexity and produce succinct and reliable software [45]. Proponents for multi-paradigm programming languages claim that no single paradigm is suited for dealing with all the possible scenarios encountered in complex programs and both are necessary to allow for flexible and high-quality programs [5, 46]. Many have claimed that Scala's multi-paradigm aspect makes it superior to Java [5, 42, 47].

Since Scala supports both the FP and OOP paradigms, it is extremely flexible and the advantages of each style can be applied to specific problems. For example, functional programming is beneficial for concurrency because variables are immutable and functions have no side effects negating the need to synchronize access to mutable state. In contrast, OOP can be used in situations that require mutability of objects.

3.2.1. Scala is Object-Oriented

Scala uses a "uniform object model" that does not distinguish between object/reference types (classes, Integer) and primitive types (int, boolean); instead everything is essentially object type. In Scala, various classes represent the types, properties, and behaviour of objects:

- **Traits**: Similar to Java's Interface, these abstract classes can be "mixed-in" and do not take constructor parameters
- **Objects**: These are classes associated with a unique instance (Singletons). In an object, the class and its instance are indistinguishable. This replaces the need for static members as in Java.
- **Case classes**: these classes are used in pattern matching which allows for the decomposition of objects, a feature largely absent in Java.

Like in Java, only single inheritance is allowed but Scala offers traits and composition of data structures through mixin. Scala also has more sophisticated typing system with generics that are more flexible and more advanced typing constructs, which provide the foundation for more type-safe design. These features have been widely claimed to allow for more customization and to be more powerful than their Java counterparts [42].

3.2.2. Scala is Functional

In Scala, every function can be assigned to variables just like values, allowing for the creation of complex operations from simple ones. Scala supports the following functional language techniques:

- **Anonymous, nested functions**: In Scala, functions with no name and nested functions can be defined.
- **Higher-order and curried functions**: All functions are objects that can be passed to other functions, allowing for the creation of "higher-order" functions which take functions as parameters. It also allows for currying, the transformation of a function of multiple parameters into a function which returns a function of one parameter each time (itself a higher-order function).
- **Closures**: "Functions can use variables that are outside of the definition scope of the function, not defined as local variables or passed as arguments." This feature simplifies the development of domain-specific languages and control abstractions [48].

Java (up until version 7) has not been a functional language, but instead emulated the behaviour of lambda expressions with anonymous inner classes (AIC). For example, adding a listener to a JComponent involved the creation of a class implementing the particular listener's interface and overriding the implemented methods. Refactoring Java code to use functional features will make the code more succinct and readable. It is claimed that Scala's functional paradigm constructs allows for clearer and more readable code, simplifies difficult tasks and testing, reduces complexity, and achieves the same task in less LOC compared to Java [42].

3.1. Scala's Benefits

3.1.1. Scala's Interoperability

Scala runs on the JVM, is completely interoperable with the Java programming environment [49]. It is also interoperable with C#, but only Java is discussed here as it is related to the topic presented by this paper. Byte code originated from a Java source can be easily invoked from Scala and similarly Scala code can be invoked from Java. Using integrated Java and Scala code is therefore possible, providing greater flexibility since Scala seamlessly integrates the Java libraries and classes. Because Java is still dominant in the market and used frequently, Scala's compatibility is extremely helpful in extending and improving current Java applications.

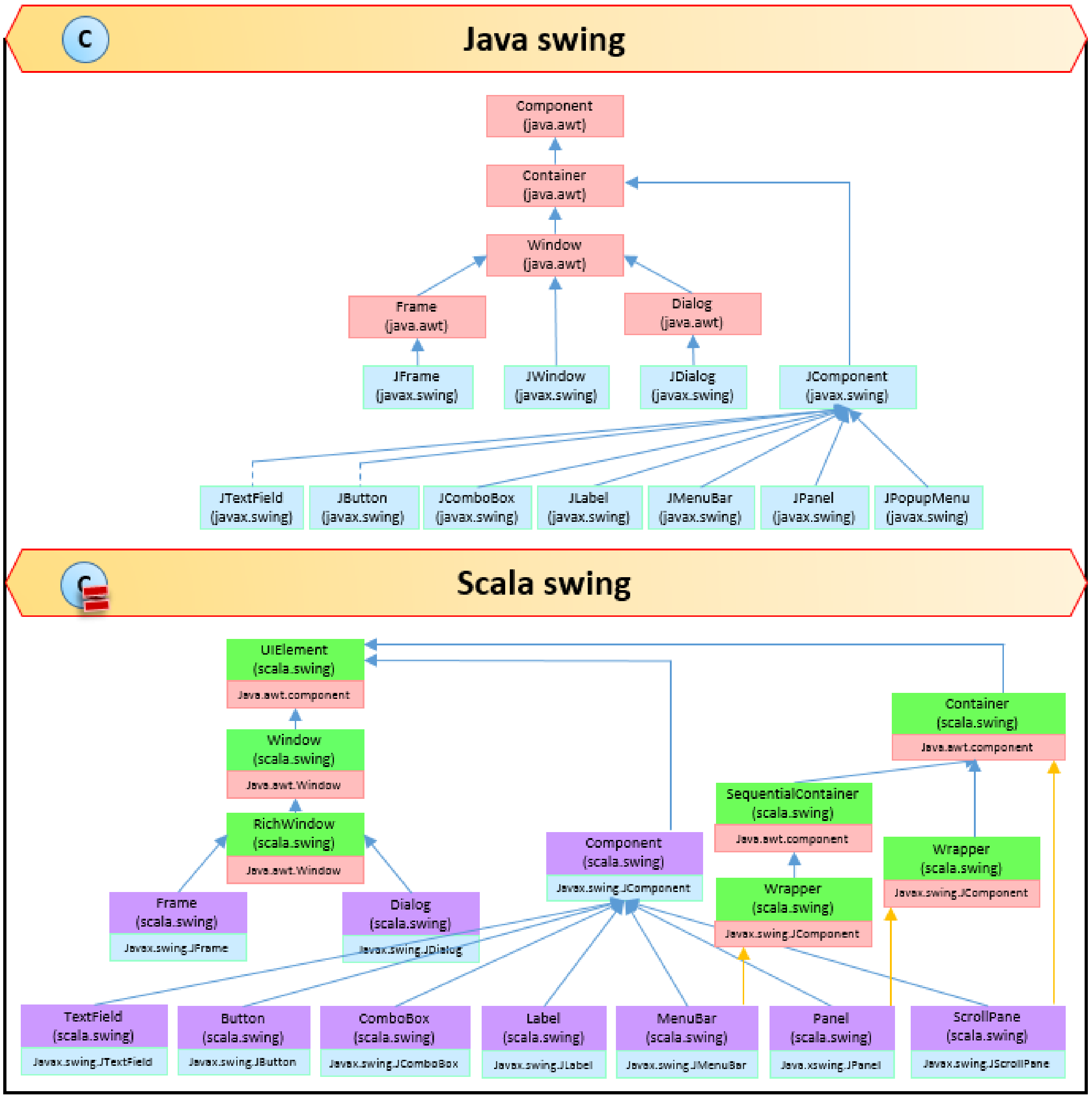
3.1.2. Scala is More Compact

Finally, Scala programs are claimed to be more compact: "a typical Scala program should have about half of the number of lines of the same program written in Java" [5]. This reduction in the lines of code (LOC) is said to be mostly due to control abstractions that avoid duplication, type inference, more efficient structures in its standard library, a simplified inheritance system, and optional semicolons [5, 48]. Consequently, the creators claim that Scala is less error-prone than Java since fewer LOC implies fewer possible places for defects.

3.1.3. Scala Supports Better Scalability

Scalability is another advantage of Scala claimed by its creators, due mostly to its ability to reduce complexity of problems with more concise high functionality code [50]. Its support for interoperability also allows for the programmer to use a combination of unique methods which will also increase Scalability [48]. With this feature, Scala can easily be used to effectively increase Scalability of Java programs and makes it ideal for small and simple to large and sophisticated applications.

3.1. Scala GUI Platform vs Java GUI Platform



Java swing
Component (java.awt)
Container (java.awt)
Window (java.awt)
Frame (java.awt)
Dialog (java.awt)
JFrame (javax.swing)
JWindow (javax.swing)
JDialog (javax.swing)
JComponent (javax.swing)
JTextField (javax.swing)
JButton (javax.swing)
JComboBox (javax.swing)
JLabel (javax.swing)
JMenuBar (javax.swing)
JPanel (javax.swing)
JPopupMenu (javax.swing)
Scala swing
UIElement (scala.swing) Java.awt.component
Window (scala.swing) Java.awt.Window
RichWindow (scala.swing) Java.awt.Window
Frame (scala.swing) Javax.swing.JFrame
Dialog (scala.swing) Javax.swing.JDialog
Component (scala.swing) Javax.swing.JComponent
Container (scala.swing) Java.awt.component
SequentialContainer (scala.swing) Java.awt.component
Wrapper (scala.swing) Javax.swing.JComponent
Wrapper (scala.swing) Javax.swing.JComponent
TextField (scala.swing) Javax.swing.JTextField
Button (scala.swing) Javax.swing.JButton
ComboBox (scala.swing) Javax.swing.JComboBox
Label (scala.swing) Javax.swing.JLabel
MenuBar (scala.swing) Javax.swing.JMenuBar
Panel (scala.swing) Java.xswing.JPanel
ScrollPane (scala.swing) Javax.swing.JScrollPane

Figure 3-1 Hierarchy of Scala and Java Swing

The following colour-coding is used in the Java Swing hierarchy: class in red box: class belongs to java.awt package, class in blue box: class belongs to javax.swing package

The following colour-coding is used in the Scala Swing hierarchy: red box under class: java.awt or javax.swing abstract peer member, blue box under class: javax.swing concrete peer member, class in green box: scala.swing class with javax.awt abstract peer member, class in purple box: scala.swing class with javax.swing concrete peer member. The blue arrows represent inheritance using the "extends" key word (traits or classes) while the orange arrows represent inheritance using the "with" key word (traits).

The original GUI platform and library for Java was Abstract Windowing Toolkit (AWT), which made direct calls to the operating system or windowing environment for building elements of the GUI. However the elements in the AWT included only elements that were present in all platforms (platform dependent), and because the elements were drawn by the underlying platform, the appearance would change depending on the platform. Therefore, the Swing GUI library was created for Java, which works by making calls to AWT. Because Swing doesn't contain any platform-specific (native) code, GUI elements could be rendered in different ways so there would be a uniform "look" across different platforms [51].

The library used for writing Scala GUI comes from the Scala.swing package, which is a wrapper for Java's Swing library (the javax.swing package which itself sits on top of the java.awt package) and makes calls to it. A summary of the hierarchy of Java swing components is presented in Figure 3-1 top while Figure 3-1 bottom summarizes the hierarchy of Scala swing components with their respective Java AWT or Swing wrapper classes. Just like Java swing, all components in Scala inherit from the root class `Component`, however unlike in Java `Component` does not descend from `Container`.

There is a Scala wrapper class for almost all JComponents and each has the same name as the Java class but without the initial 'J' (for example "JButton" becomes "Button"). Methods in the associated JComponent can be called from a Scala Component by using the `peer` method. Note that some classes have peer as an abstract value member, so this member cannot be directly used until a concrete implementation has been given (for example in Figure 3-1 (bottom), `peer` is implemented in `Frame` and `Dialogue` after inheriting the abstract member from `RichWindow`).

Figure 3-1 top also shows Java/Swing's window hierarchy with AWT's `Frame` and `Dialog` both sharing the base class `Window`. However the `JFrame` and `JDialog` classes do not extend `JWindow` even though they share common functionality not present in AWT. Scala provides better design of the window hierarchy as seen in Figure 3-1 bottom where `Frame` and `Dialog` have `Window` as a common ancestor, with common wrapper code refactored into `RichWindow`.

Scala GUIs use the same concepts as Java, using the benefits of Java's Swing library but with simpler Scala syntax which makes code more manageable. To build a complex GUI application, both Scala and Java use the idea of a containment hierarchy, which builds GUI by adding components and nesting them within each other. In Java, all JComponents extend from `Container` and can all hold child JComponents within them. The Scala `Container` is a trait that extracts a common interface to provide components, menus, and windows with the ability to hold child components. For example, `MenuBar` extends `SequentialContainer.Wrapper` which extends `Container` and Panel extends `Container.Wrapper` (Figure 3-1) therefore both possess the ability to nest child components. For both Scala and Java, a top-level component like a panel is used to hold an arbitrary number of child components inside it, to achieve the overall GUI layout in the end.

The laying out of the child components is governed by layout rules specified in the layout manager. It determines the behaviour of the layout when resized, and determines the size and position of the components within a container. In Java, the layout manager is an object implementing the `LayoutManager` interface and can be set by the user while in Scala one of the layout containers is used (e.g. `BorderPanel` combines a `Panel` with the border layout features). For both Java and Scala, users can specify options, which make the layout rules customizable to some extent.

Section 6.1 continues the introduction of general Scala and Scala swing with practical examples from the ALE source code, which includes these topics: Java vs Scala event handling – including pattern matching (Section 6.1.2), partial functions (Section 6.1.3.1), Scala collections (Section 6.1.4), and Scala's wrapper system – including traits (Section 6.1.5).

Chapter 4. The Auckland Layout Model (ALM)

The Auckland Layout Model (ALM) is the constraint-based layout engine used for the Auckland Layout Editor (ALE). This section provides an overview of the advantages of ALM and introduces important ALM classes.

4.1. Overview

ALM forms its layout specification using a set of constraints based on linear algebra while the optimal layout is calculated using linear or quadratic programming. Since linear programming is difficult to understand, ALE simplifies GUI building by offering four levels of abstraction: 1) linear constraints/linear programming, 2) soft constraints, 3) areas, 4) rows and columns. It separates different parts of the layout specification into different modules, allowing each to be managed separately with the potential to be recombined later. In ALM, variables used in a constraint are vertical or horizontal lines known as X and Y tabstops (or tabs), respectively. They are created by extending every edge of each rectangular area and define the x- and y- coordinates within the GUI coordinate system, forming a type of grid layout [1].

4.2. Classes

Figure 4-1 presents a class diagram of the important ALM classes in the Java version. All classes are described below are situated in the `alm` package, except for the `Constraint`, `Summand`, and `Variable` which are in the `linsolve` package.

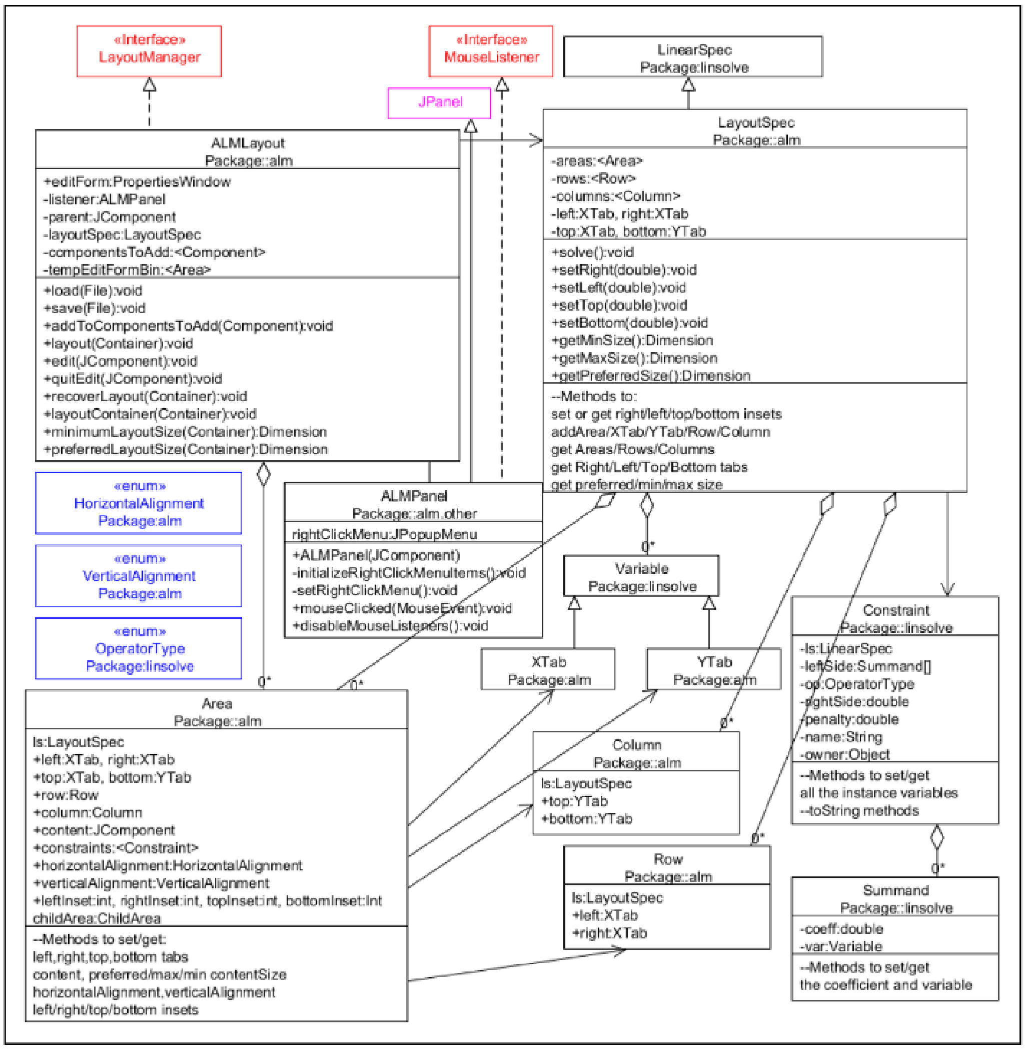

«Interface» LayoutManager
«Interface» MouseListener
LinearSpec Package:linsolve
JPanel
LayoutSpec Package::alm
-areas:<Area>
-rows:<Row>
-columns:<Column>
-left:XTab, right:XTab
-top:XTab, bottom:YTab
+solve():void
+setRight(double):void
+setLeft(double):void
+setTop(double):void
+setBottom(double):void
+getMinSize():Dimension
+getMaxSize():Dimension
+getPreferredSize():Dimension
--Methods to:
set or get right/left/top/bottom insets
addArea/XTab/YTab/Row/Column
get Areas/Rows/Columns
get Right/Left/Top/Bottom tabs
get preferred/min/max size
ALMLayout Package::alm
+editForm:PropertiesWindow
-listener:ALMPanel
-parent:JComponent
-layoutSpec:LayoutSpec
-componentsToAdd:<Component>
-tempEditFormBin:<Area>
+load(File):void
+save(File):void
+addToComponentsToAdd(Component):void
+layout(Container):void
+edit(JComponent):void
+quitEdit(JComponent):void
+recoverLayout(Container):void
+layoutContainer(Container):void
+minimumLayoutSize(Container):Dimension
+preferredLayoutSize(Container):Dimension
«enum» HorizontalAlignment Package:alm
«enum» VerticalAlignment Package:alm
«enum» OperatorType Package:linsolve
ALMPanel Package::alm.other
rightClickMenu:JPopupMenu
+ALMPanel(JComponent)
-initializeRightClickMenuItems():void
-setRightClickMenu():void
+mouseClicked(MouseEvent):void
+disableMouseListeners():void
Variable Package:linsolve
XTab Package:alm
YTab Package:alm
Constraint Package::linsolve
-ls:LinearSpec
-leftSide:Summand[]
-op:OperatorType
-rightSide:double
-penalty:double
-name:String
-owner:Object
--Methods to set/get all the instance variables
--toString methods
Area Package::alm
ls:LayoutSpec
+left:XTab, right:XTab
+top:XTab, bottom:YTab
+row:Row
+column:Column
+content:JComponent
+constraints:<Constraint>
+horizontalAlignment:HorizontalAlignment
+verticalAlignment:VerticalAlignment
+leftInset:int, rightInset:int, topInset:int, bottomInset:Int
childArea:ChildArea
--Methods to set/get:
left,right,top,bottom tabs
content, preferred/max/min contentSize
horizontalAlignment,verticalAlignment
left/right/top/bottom insets
Column Package::alm
ls:LayoutSpec
+top:YTab
+bottom:YTab
Row Package::alm
ls:LayoutSpec
+left:XTab
+right:XTab
Summand Package::linsolve
-coeff:double
-var:Variable
--Methods to set/get the coefficient and variable
0*
0*
0*
0*
0*
0*

Figure 4-1 Class diagram of ALM classes in ALE Java version

4.2.1. ALMLayout

This class implements `LayoutManager`, provides the methods for specifying the rules of the layout, and manages insets, areas, rows and columns. It contains lists to store components added to the GUI testing window and the discarded components in the bin. Methods include those to initialize layout (in which a new instance of `LayoutSpec` is created; each `ALMLayout` is associated with its own `LayoutSpec`), to calculate and set the layout, to switch the mode to editing mode, and other general methods associated with recovering and changing the layout (most accessed from the `LayoutSpec` instance).

4.2.2. LayoutSpec

Contains information for the layout specification: the class contains lists to store the columns, rows, and areas in the specification. It also contains `X/YTab` instance variables define the top, bottom, left and right borders of the component being edited. There are also methods for getting/setting areas, rows, columns, XTab, YTabs, insets, and sizes.

4.2.3. ALMPanel

For the purposes of this study, only its functionality for providing a right click menu for the test GUI is described here. When the user switches to editing mode, this class adds a `MouseListener` to the JComponent containing the test GUI components (for example the JPanel in the TestEdit1 window) and also defines a method to show a popup menu at the position of a right click on it.

4.2.1. X/YTab

`XTab` and `YTab` extend the `Variable` superclass (since they are variables used in constraints) and represent the virtual gridlines grid lines, which define the boundaries of areas, child areas, and inset tabs.

4.2.2. Constraint

At its core, ALM uses linear constraints (linear equalities and inequalities) and a linear objective function (a linear combination of variables/tabstops and their constant coefficients) to specify its layout. The first type of constraint, called absolute constraint, is defined by setting the width or height between two tabs to a particular value, or fixing tabs at particular positions. The second type, called relative constraint, defines tabs at positions relative to other tabs. Resizing the window causes the adjustment of the tabs for relative constraints while they are unaffected for absolute constraints.

Flexible constraints known as soft constraints can be specified which helps prevent over-constraint specifications. Unlike hard constraints, which are satisfied strictly, soft constraints, may be violated if circumstances do not permit their satisfaction.

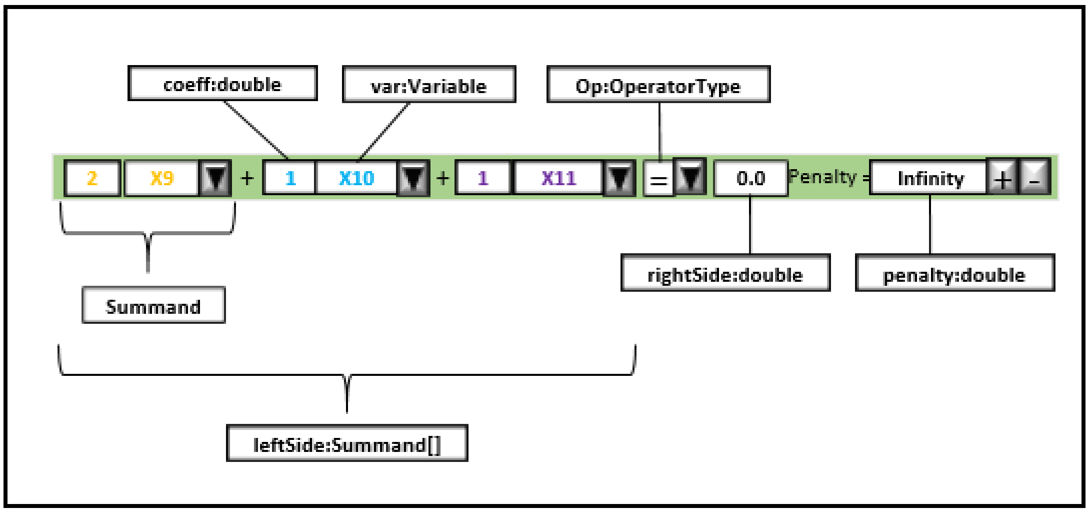


Figure 4-2 Overview of a constraint and illustration of how instance variables in the constraint class correspond to a constraint used in the layout specification

The `Constraint` class contains instance variables that correspond to each property in a constraint, as shown by Figure 4-2.

- `summand` is the combination of a coefficient and variable
- `leftSide` is an array containing all the summands in the constraint
- `op` enum represents the operator type and is either = (for linear equality), <= or >= (for linear inequalities)
- `rightSide` (double) is the “right side” of the equation
- `penalty` (double) for if the constraint gets violated

4.2.3. Area

A rectangular area (which may contain components or other graphical elements) is bound by a pair of X and Y tabstops. The `Area` class contains all the information needed in to define a rectangular area including:

- preferred, minimum, maximum size
- its boundaries (left, right XTabs and top, bottom YTabs)
- the row/column it belongs to
- its content (the JComponent it contains)
- its horizontal/vertical alignment (enums: left, right, center, fill)
- its child area (null if it doesn’t contain one, used to show inset tabs)
- left/right/top/bottom inset

4.2.4. Row/Column

Rows and columns are represented either by a y interval (pair of x tabstops) or x interval (pair of y tabstops), respectively, and can contain several areas. These features are not yet implemented and these classes are not yet used.

Chapter 5. The Auckland Layout Editor (ALE)

This Chapter gives an overview of the Auckland Layout Editor (ALE) [2], i.e. its general concepts and its Java code base at the start of this project, i.e. before any changes were made. It describes its various classes, the steps following the initialization of the editing mode, the containment hierarchy of its GUI, and its edit operations and other features.

5.1. Overview

ALE is a WYSIWYG GUI builder, which allows the end-user to manipulate and customize GUI dynamically. ALM has the potential to support many edit operations while maintaining all constraints in the specification and supporting the non-overlapping of components in the layout. Editing mode is initiated by calling the method `edit` defined in `LayoutManager` which converts the components in the GUI into bitmap images so that their appearance is maintained while removing their functionality. This is accompanied by the appearance of a properties window that contains editable components for editing the specification of the GUI layout. Only the area and constraints modes are discussed in this study since the others have not been implemented yet.

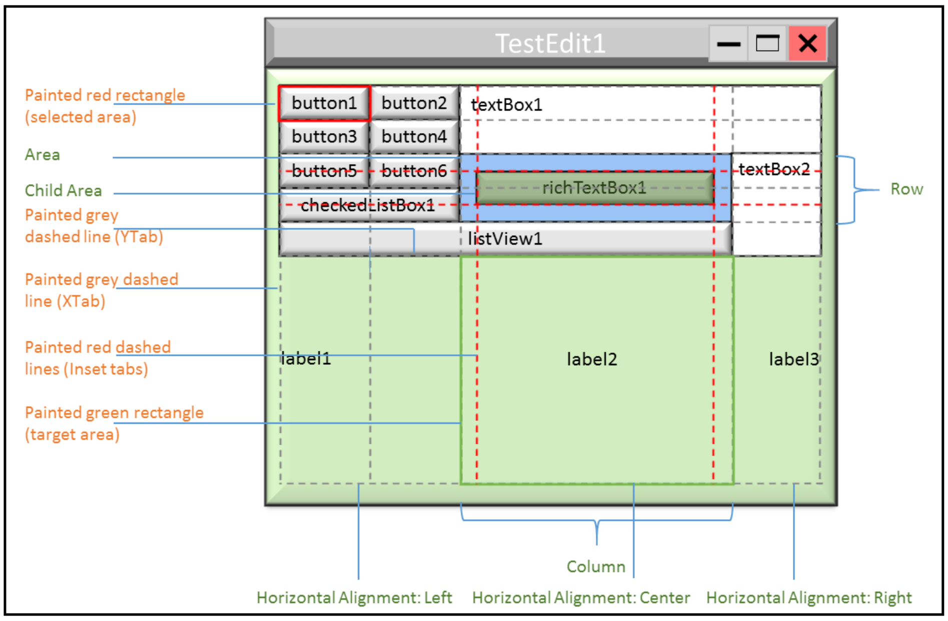

TestEdit1
Painted red rectangle (selected area)
Area
Child Area
Painted grey dashed line (YTab)
Painted grey dashed line (XTab)
Painted red dashed lines (Inset tabs)
Painted green rectangle (target area)
button1
button2
button3
button4
button5
button6
textBox1
textBox2
richTextBox1
checkedListBox1
listView1
label1
label2
label3
Row
Column
Horizontal Alignment: Left
Horizontal Alignment: Center
Horizontal Alignment: Right

Figure 5-1 A pictorial representation of the appearance of the GUI window/ testing window in area mode

Figure 5-1 shows a pictorial representation of the testing window called TestEdit1 (mimicking the actual appearance in Figure 1-1); these pictorial representations are used from now on instead of screen shots. Note that `TestEdit1` is simply an example using the ALM to place 14 components onto a JPanel. Other examples exist and more could be created but for simplicity only the TestEdit1 example is used throughout this report.

The figure illustrates the ALM concepts (green labels) as well as how ALE displays them (orange labels). Label1, Label2, and Label3 have left, centre, and right horizontal alignment property, respectively. The figure also label a row (pair of y-tabs) and a column (pair of x-tabs) which can be used to define, rearrange and resize areas easily without directly referencing tabs, by employing the table metaphor. In the area mode, the area selected by the mouse is highlighted by a red rectangle while the target area is highlighted by a green rectangle. In both modes, the usable X/YTabs are represented by grey dashed lines and inset X/YTabs by red dashed lines.

An inset is the space between an area and its content. A child area is created when the end-user either requests insets or special alignment on a particular area, which means the control is smaller than the borders of an area. In Figure 5-1 the area highlighted in blue contains a richTextBox1 with left, right, top, bottom inset values of ten. A child area is created to with X/YTabs corresponding to the left, right, top, bottom borders of richTextBox1. Unlike the usable X/YTabs displayed as grey dashed lines, these inset X/YTabs cannot be used as variables in a constraint.

5.2. Classes

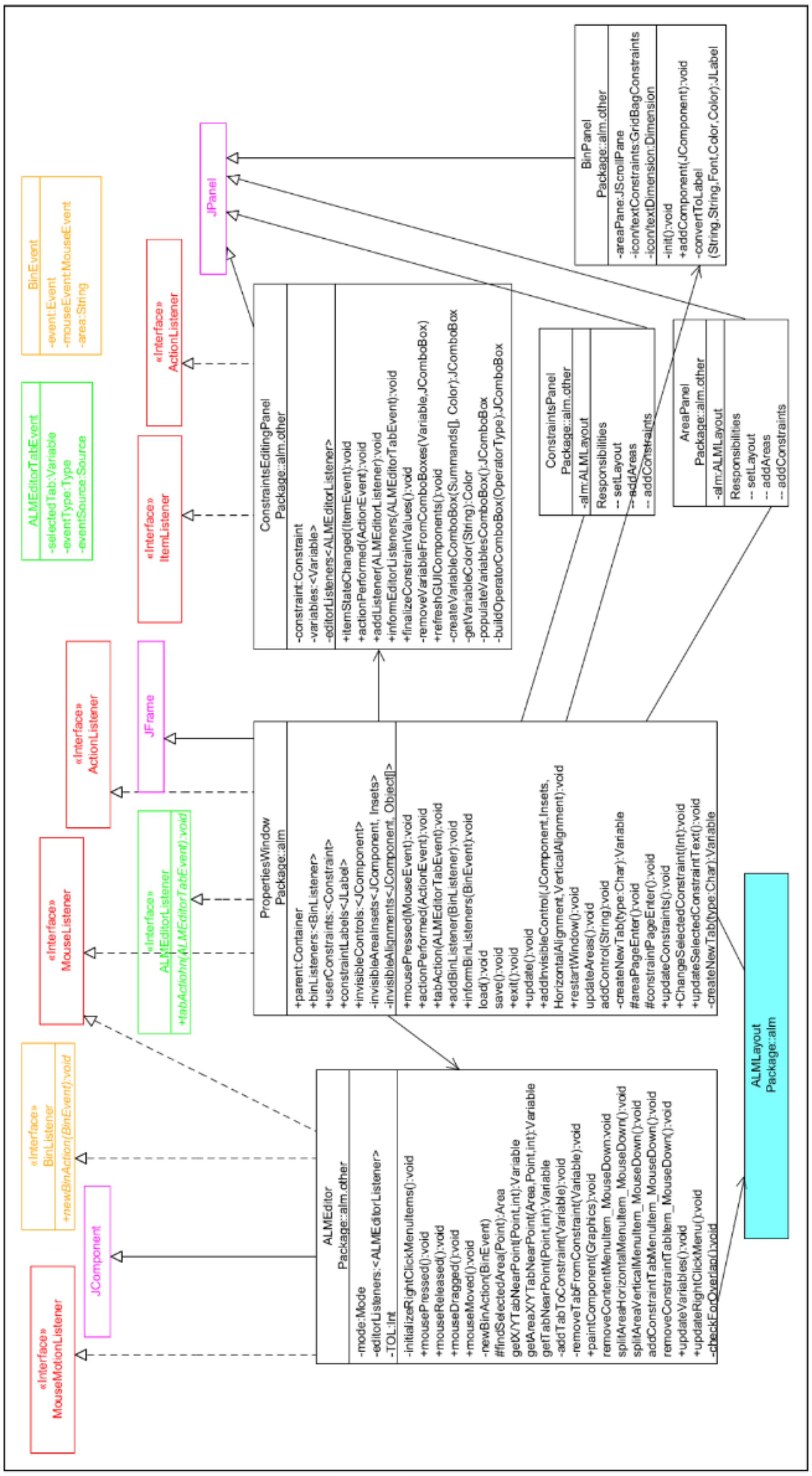


Figure 5-2 Class diagram of ALE, the old implementation

Figure 5-2 presents a class diagram of the important classes associated with the editor. All classes are described below are situated in the `alm.other` package, except for `PropertiesWindow` which is in the `alm` package.

5.2.1. ALMEditor

This class is the editing canvas which is layered on top of the testing window in the editing mode. Its main purpose is to 1) detect mouse clicks and movements and either respond by calling methods within itself or relay the information to `PropertiesWindow` and 2) to replace the functional GUI components with bitmap images. Most of the functionality for editing is defined in this class, the rest is defined in `PropertiesWindow.`

- It has an enum inner class for the mode enum variables
- It initializes the popup menu and its menu items and keeps track of what should be in the popup menu in each mode, and for the constraint mode, the items change dynamically depending on the position of the right click
- It implements `MouseListener`, and `MouseMotionListener`. The `MousePressed/Released/Moved/Dragged` event handler methods contains code for the detection of the mouse on the canvas to provide information for the edit operations.
- It defines the `paintComponent` method which paints the XTabs, YTabs, inset lines (if show inset tabs was selected), the selected area with red border, destination area with green border, and constraint lines. It also goes through all the areas in the layout specification and uses `BufferedImage` to draw the component in each area in the canvas
- It implements the `BinListener` interface and provides an implementation of the `newBinAction` method

5.2.2. PropertiesWindow

This class contains the majority of the code in the properties window that appears when editing mode is started. It is the mediator for most of the activity, containing many components and methods and has references to most of the other classes (`ALMEditor`, `ALMLayout`, `AreaPanel`, `ConstraintsPanel`, and `BinPanel`).

- It has an instance of the `ALMLayout` associated with the GUI currently being edited
- It sets up a menu bar with load, save (load and save a particular layout) and exit options.
- It contains all the variables for child components in the properties window, including those in the areas panel and constraints panel
- It contains all the methods relevant to the features and operations performed by the properties window’s child components.
- Event listeners are added to all the interactive components
- It implements the `ALMEditorListener` interface and provides an implementation of the `newTabAction` method
- It has a method for updating the components associated with editing the areas and another method for updating the components associated with editing the constraints

5.2.3. AreaPanel and ConstraintsPanel

`AreaPanel` and `ConstraintsPanel` define the layout for the area panel and constraints panel, respectively. The `AreaPanel` and `ConstraintsPanel` classes have their own instance of `ALMLayout`, which is used to set up the GUI layout for their panel in the properties window. The GUI components themselves are instance variables in `PropertiesWindow`, so there is an extra step needed to retrieve them when adding each component to the area/constraints panel.

5.2.4. ConstraintsEditingPanel

A class that contains the components and methods associated with the single line that becomes editable when it is clicked in the constraints panel.

- It contains a reference to the constraint that it is editing
- It uses flow layout to add each of the editable components shown in Figure 5-3
- It contains a method to change the text in the components to reflect any new changes made to the editable components

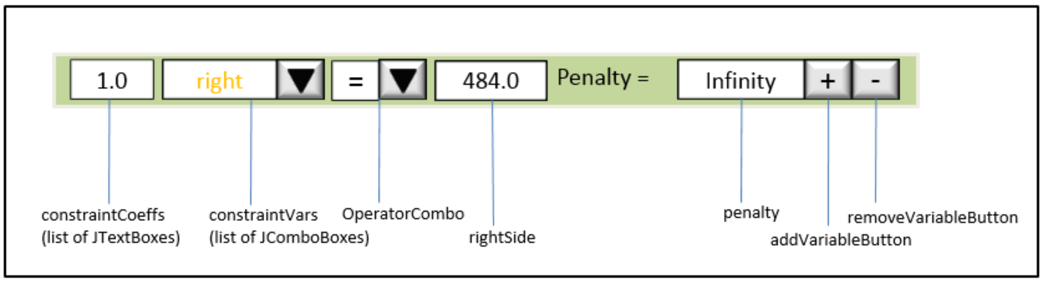


Figure 5-3 The editable components in ConstraintEditingPanel class

5.2.5. BinPanel

This JPanel uses a gridBagLayout layout manager to layout its bin components, however the list containing the bin items are in `PropertiesWindow`. It contains two methods which are called through the `PropertiesWindow` class when the user removes a GUI component from the window: the `convertToLabel` method which converts the bin item to a JLabel for display, and the `addComponent` method which makes the JLabel visible on the bin panel.

5.3. Switching a GUI into Editing mode

The three figures below (Figure 5-4, Figure 5-5, and Figure 5-6) illustrate the sequence of steps that occur after editing mode is started all the up until everything is displayed correctly in both the testing window and properties window. Each box with a title represents a class. Within the class, the references to other classes is represented as a purple rounded box while its variable name is in purple below it. The important methods and constructors are labelled with red circles containing 'm' and 'c' respectively. Instantiation of each class is marked with a red circle containing the down arrow. The grey circled numbers represent the order in which the class was instantiated. For example, `PropertiesWindow` is the third class to be instantiated: it is labelled "3". Lastly, the containment hierarchy at each step is illustrated showing what components are added to the container class.

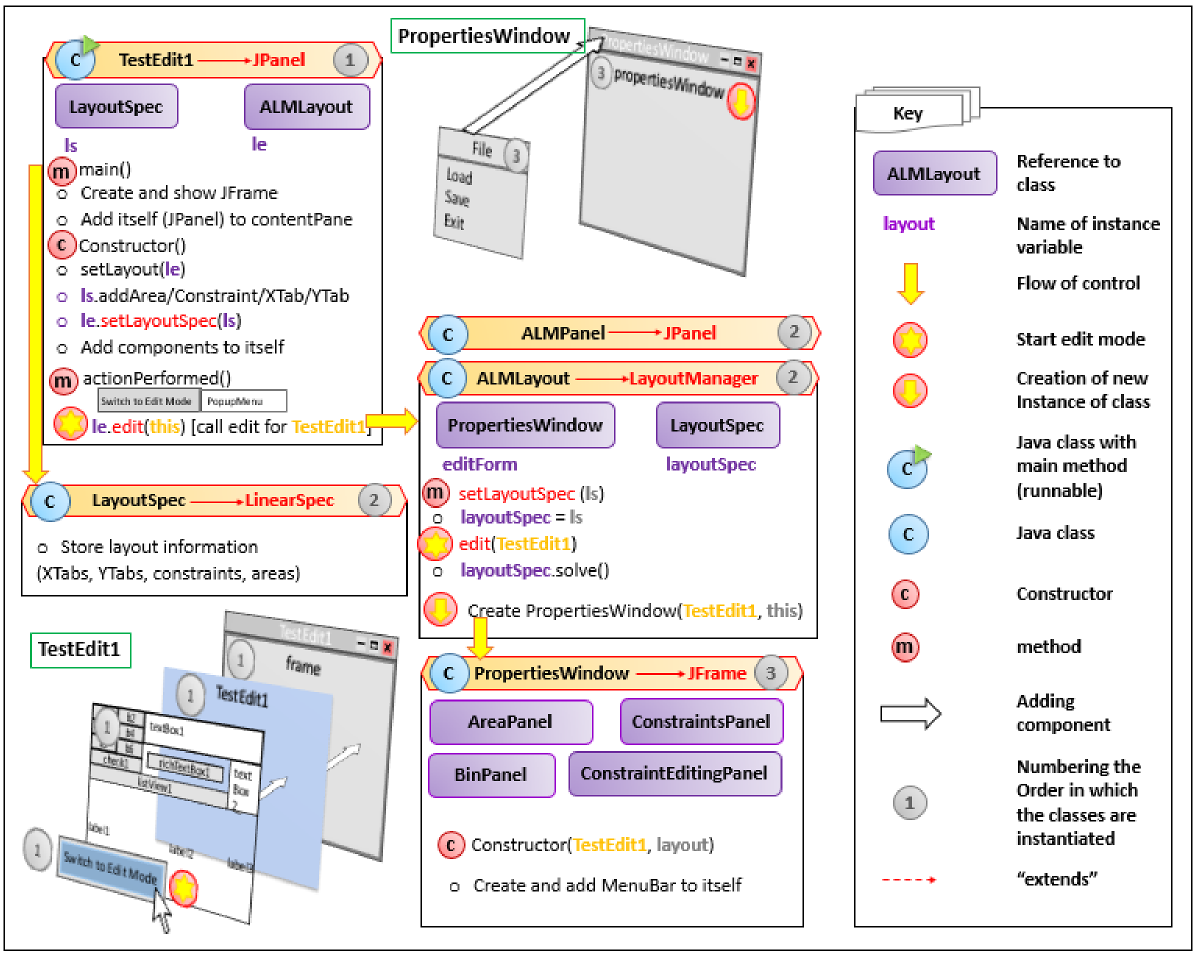
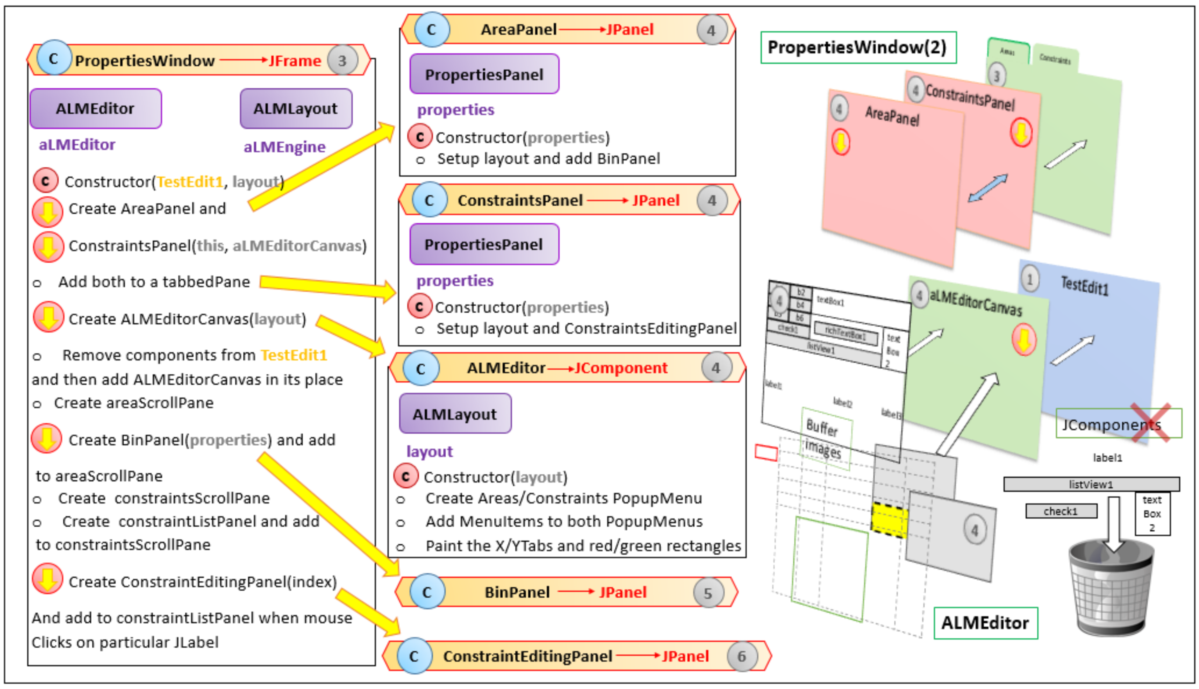


Figure 5-4 Initialization of the editing mode by clicking the "Switch to Edit mode" menu item and the instantiation of LayoutSpec, ALMLayout and PropertiesWindow classes.



PropertiesWindow → JFrame 3
ALMEditor
ALMLayout
aLMEditor
aLMEngine
Constructor(TestEdit1, layout)
Create AreaPanel and
ConstraintsPanel(this, aLMEditorCanvas)
Add both to a tabbedPane
Create ALMEditorCanvas(layout)
Remove components from TestEdit1 and then add ALMEditorCanvas in its place
Create areaScrollPane
Create BinPanel(properties) and add to areaScrollPane
Create constraintsScrollPane
Create constraintListPanel and add to constraintsScrollPane
Create ConstraintEditingPanel(index)
And add to constraintListPanel when mouse Clicks on particular JLabel
AreaPanel → JPanel 4
PropertiesPanel
properties
Constructor(properties)
Setup layout and add BinPanel
ConstraintsPanel → JPanel 4
PropertiesPanel
properties
Constructor(properties)
Setup layout and ConstraintsEditingPanel
ALMEditor → JComponent 4
ALMLayout
layout
Constructor(layout)
Create Areas/Constraints PopupMenu
Add MenuItems to both PopupMenus
Paint the X/YTabs and red/green rectangles
BinPanel → JPanel 5
ConstraintEditingPanel → JPanel 6
PropertiesWindow(2)
AreaPanel
ConstraintsPanel
aLMEditorCanvas
TestEdit1
JComponents
Buffer images
listView1
check1
text Box 2
label1
ALMEditor

Figure 5-5 Continuing the initialization of the editing mode: the activity of the PropertiesWindow, and showing the instantiation of AreaPanel, ConstraintsPanel, ALMEditor, BinPanel and ConstraintEditingPanel classes

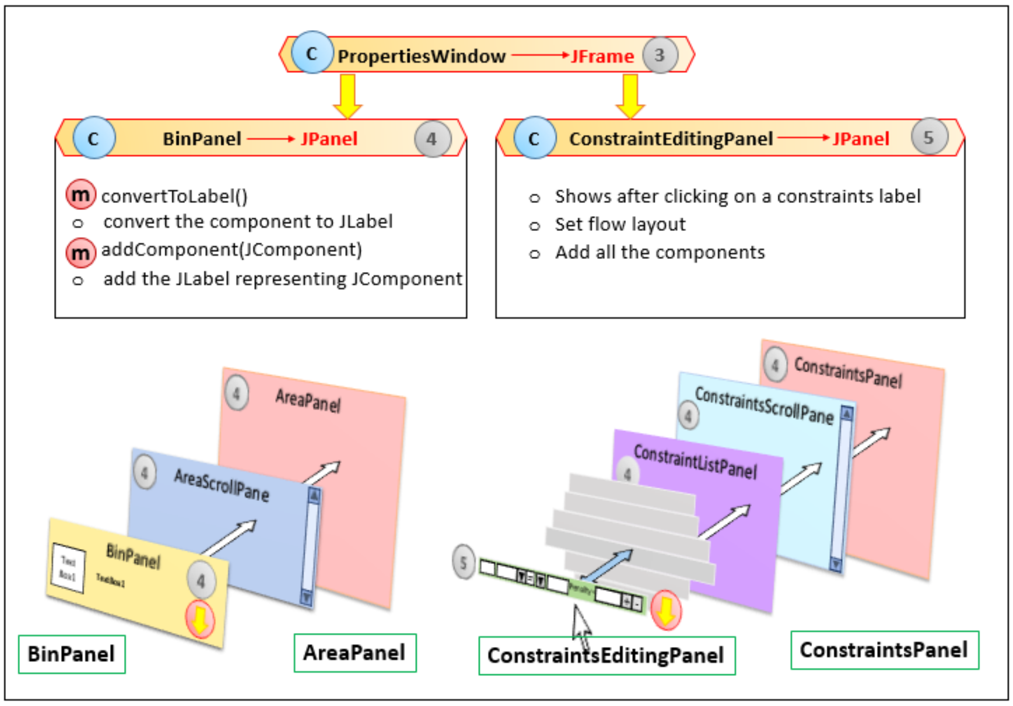


Figure 5-6 final of the initialization of the editing mode: the activities of BinPanel and ConstraintEditingPanel

1) TestEdit1 class

This class contains a main method, when executed, will display a GUI example to the end-user. First the ALMLayout and LayoutSpec classes are created, then the example components are added to itself (which is a JPanel), and finally the GUI example is added to and displayed in a JFrame.

In the constructor, ALMLayout and LayoutSpec are instantiated (variables named le and ls). setLayout method defined in ALMLayout is called and ls is passed as argument and becomes set as an instance variable in ALMLayout. Now this GUI is associated with one instance of ALMLayout which manages the layout of the components using information stored in one instance of LayoutSpec. Methods for adding an X/YTab, adding an area, and adding constraints are called so that JButtons, JLabels, and JTextBoxes are set up in their correct positions. These X/YTabs, areas and constraints now stored in LayoutSpec and can be accessed later on. setLayoutSpec is called with the instance of the LayoutSpec passed as argument.

Also, an instance of the ALMPanel is created which registers MouseListener on the JPanel. The testing window now has the reaction installed to display a popup menu containing the menu item “switch to editing mode” whenever the user right clicks on it. Lastly, an instance of the class itself (the newly set-up JPanel with 14 JComponents and popup menu) is added to a newly created JFrame.

Clicking on the menu item will start the editing process by calling the edit method from the ALMLayout class.

2) LayoutSpec class and ALMLayout class

One instance of LayoutSpec is created for the particular test window, it is used to store layout information containing lists of XTabs, YTabs, constraints and areas used to specify and modify the layout.

One instance of ALMLayout is created for the layout in the TestEdit1 example, it contains and stores the instance of LayoutSpec, and also stores the instance of PropertiesWindow. It also contains the edit method which is called and passed the JComponent which is being edited (in this example it is the TestEdit1 JPanel). Within the edit method, an instance of PropertiesWindow is created.

3) Properties Window class

Within the constructor, one instance each of `AreaPanel` and `ConstraintsPanel` are created and added to a newly instantiated tabbedPane. It also makes the two JPanels to be scrollable.

Also, an instance of `ALMEditorCanvas` is created, the size set to be the same as the `TestEdit1` JPanel. Following this, all the JComponents contained in the `TestEdit1` JPanel are removed and `ALMEditorCanvas` (which extends JComponent) is layered on the JPanel instead. Also, a menu bar containing the menu items "load", "save", and "exit" are created and added to itself.

Lastly, the `BinPanel` is instantiated and made scrollable. The `ConstraintsEditingPanel` is instantiated in the constructor but is only added to the `constraintListPanel` JPanel when a JLabel has been clicked, which replaces the selected JLabel with an instance the `ConstraintsEditingPanel` JPanel with editable components corresponding to the selected constraint.

4) ALMEditorCanvas class

Within its constructor, a popup Menu is created and added to itself. The class also overrides the `paintComponent` method which paints the all the components stored in the ALMLayout instance so the TestEdit1 window looks as if it still contains all the JComponents previously there. Furthermore, the `paintComponent` method paints the X/YTabs and red/green rectangles seen in the window.

4) AreaPanel/ConstraintsPanel

Both classes use their own instance of ALMLayout to set up the layout in their respective panels. They add the correct JComponents (including BinPanel for AreaPanel and ConstraintsEditingPanel for ConstraintsPanel) to themselves but first have to access them from `PropertiesWindow`.

4) BinPanel

It simply sets up gridBag layout and contain two methods for manipulating bin items which are used by the `PropertiesWindow`. It contains nothing at the start of the editing mode.

5) ConstraintsEditingPanel,

On clicking a particular JLabel, the `ConstraintsEditingPanel` is initialized and displayed at the position of the JLabel. At this moment, all its components are initialized and added to itself. Thus, the selected JLabel is replaced by a line of JComponents which allow for the editing of the selected constraint.

5.4. GUI Containment Hierarchy

The section below contains diagrams which illustrate the containment hierarchy for the TestEdit1 window (Figure 5-7), properties window in area mode (Figure 5-8) and properties window in constraints mode (Figure 5-9).

5.4.1. TestingEdit1

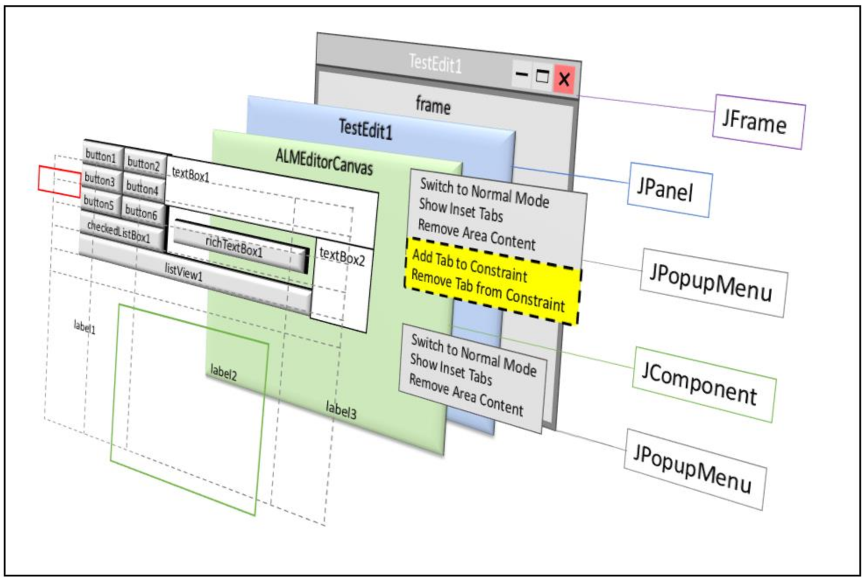


Figure 5-7 Containment hierarchy for the testing window (old implementation)

5.4.2. PropertiesWindow (Area Mode)

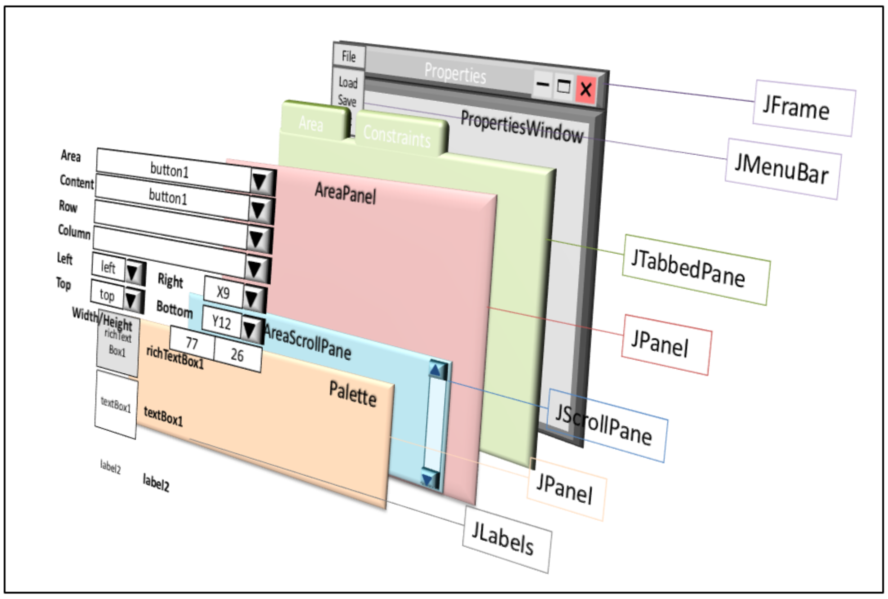


Figure 5-8 Containment hierarchy for the properties window (area mode) (old implementation)

5.4.3. PropertiesWindow (Constraints Mode)

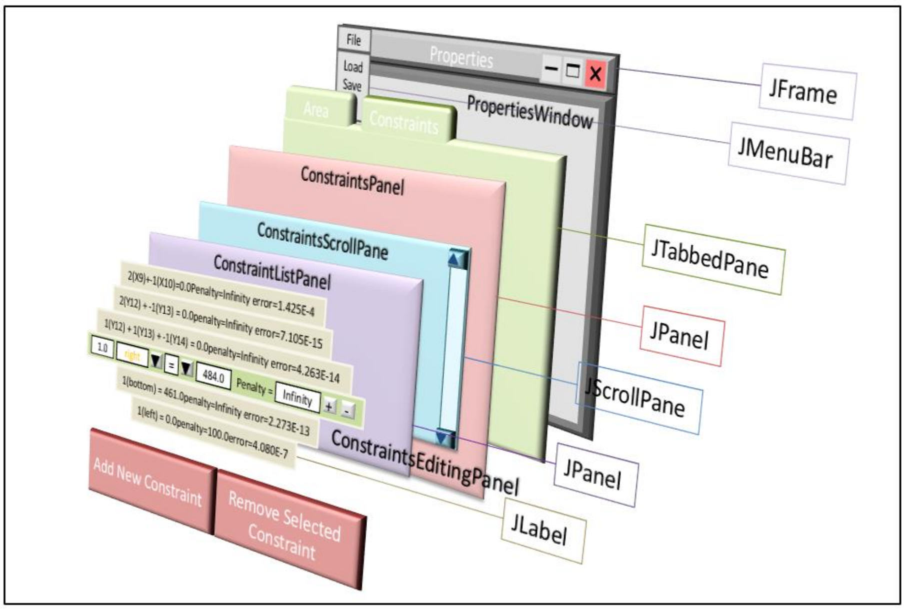


Figure 5-9 Containment hierarchy for the properties window (constraints mode) (old implementation)

5.5. General Features and Edit Operations

This section describes the edit operations and other features found in ALE Java version: those in both modes (Section 5.5.1), in area mode only (Section 5.5.2) and in constraints mode only (Section 5.5.3). Note that each is indexed by a letter: the letter is contained within a grey circle for features and in a yellow circle with the Scala logo for edit operations, indicating that these are factored out into new Scala classes in the new implementation (see Section 6.2.2.2). Edit operations are those actions which change and customize the layout, most of these were also present in the C++ version of ALE [2] , while general features are the remaining actions that an end-user can perform.

5.5.1. Features in Both Area and Constraints Modes

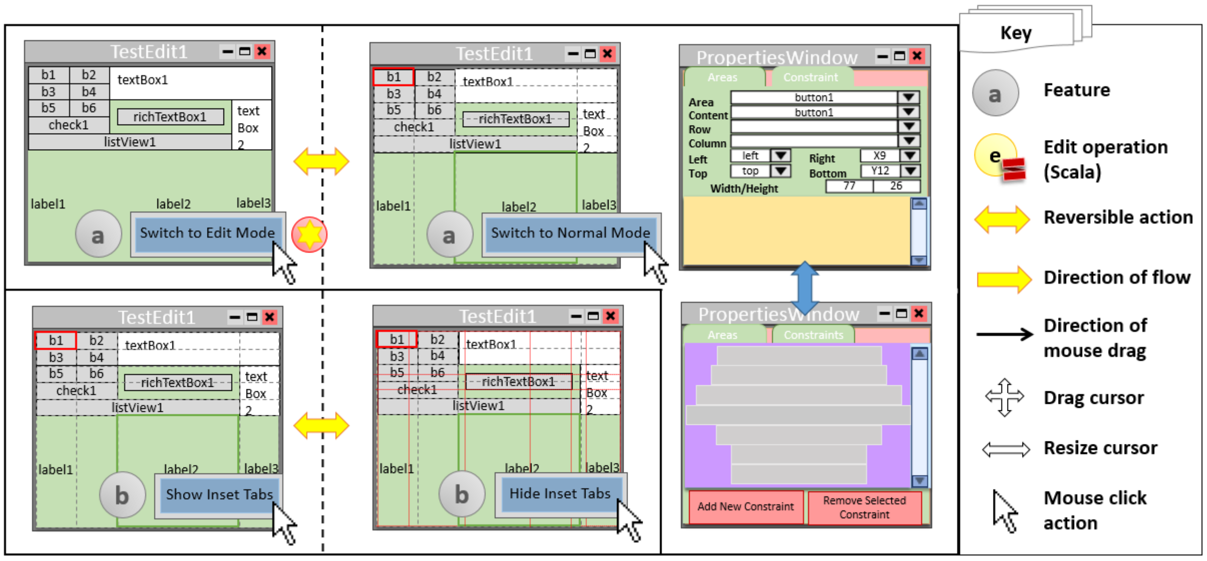


Figure 5-10 Features in both area and constraints modes

Switching between editing and normal mode (Figure 5-10a)

Note that normal mode is another term for operational mode described earlier in this report. After "Switch to Edit Mode" has been clicked in the popup menu, the properties window will become visible and subsequently the editing mode is started. It contains two tabs signifying the two different editing modes one can choose to edit the GUI in the testing window (Figure 5-10, the two-way blue arrow signifies the ability to switch between the tabs). The process for starting the editing process, including the names of the classes and the order that they are instantiated, is summarized above in Section 5.3.

In both modes, clicking on "Switch to Normal Mode" menu item in the popup menu will close the properties window and terminate the editing process and go back to normal/operational mode. All the buffered images in the testing window are changed back into JComponents corresponding to the position of their images at the end of the process. Alternatively simply exiting the properties window will achieve the same effect.

Show/hide inset tabs (Figure 5-10 b)

Clicking on "Show Inset Tabs" menu item shows the inset tabs for the JComponents in the TestEdit1 window as dashed red lines (Figure 5-10b right). Clicking "Hide Inset Tabs" menu item hides them (Figure 5-10b left).

5.5.2. Features and Edit Operations in Area Mode

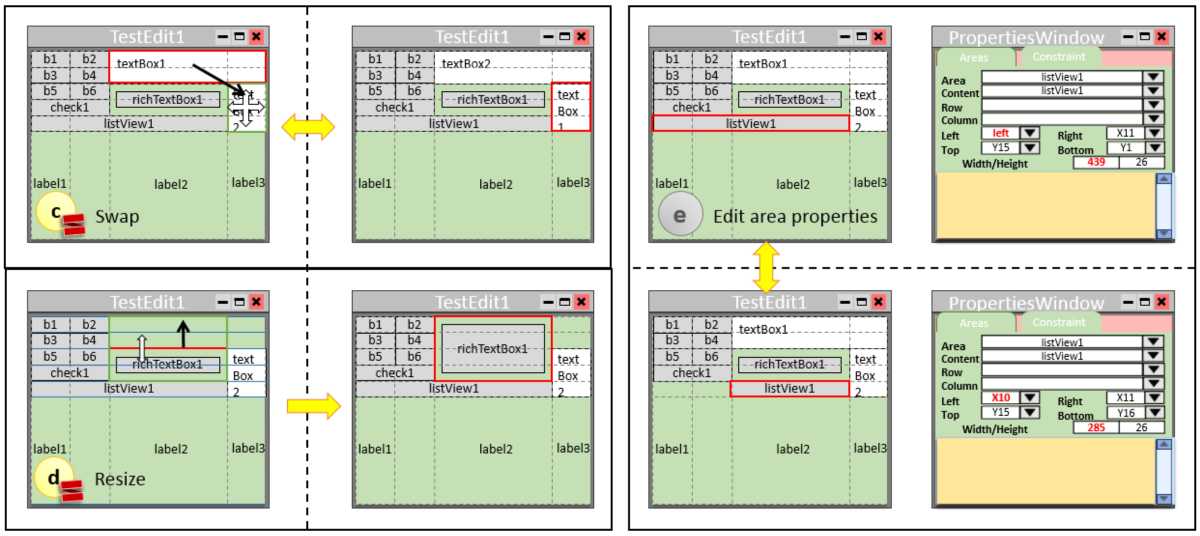


Figure 5-11 Edit operations and features in area mode part 1

Swapping areas (Figure 5-11c)

Swapping areas occurs by clicking on selecting an area (by clicking on it, for example in the figure the area containing textBox1 is selected in Figure 5-11c left), then dragging the mouse to the destination area (for example, the area containing textBox2 in the Figure 5-11c left). When the mouse is released, the two areas will swap positions (Figure 5-11c right it appears as if textBox1 has swapped position with textBox2).

Resizing an area (Figure 5-11d)

When scrolling to the border of an area, the cursor will change to a bi-directional arrow; indicating that one can drag and change the size of an area (see the white arrow in Figure 5-11d left). Also, selecting a YTab causes all the usable YTabs (i.e. not inset tabs) to change from grey dashed lines to blue solid lines to aid visualisation for resizing (the same is true if a XTab is selected). Resizing can occur provided that the end result doesn't cause overlapping of different areas (although this checking mechanism is not fully functional so some overlap does occur in certain situations). In Figure 5-11d left, the top border of the area is dragged to the top-most YTab and released (the green border shows the size of the area after release). The right image shows the resulting resized area; the occupying component (richTextBox1) appears to be resized.

Edit area operations (Figure 5-11e)

- **Area and content combo boxes:** contain the names of all the components in testing GUI. Any combo box selection will automatically update the selection in the testing window. Similarly, clicking on the listView1 button in the testing window will make the listView1 selection automatically appear in the area and content combo boxes (for example, the listView1 is selected in Figure 5-11e). Note that the area and the content which occupies the area is have the same name.
- **Row and column combo boxes:** Not yet implemented.
- **Left, right, top and bottom combo boxes:** Left and right combo boxes list all the XTabs, while top and bottom combo boxes lists all the YTabs in the layout specification. Selecting a different Tab from the left, right, top, or bottom combo boxes changes the left, right, top or bottom boundaries of an area in the testing window. (In Figure 5-11e, the top images show listView with the “left” XTab as its left boundary and the bottom images show listView with “X10”) as its left boundary).
- **Width/Height text fields:** These contain information regarding the width and height of the selected area. They are not editable but change corresponding to the changes made to an area (In Figure 5-11e, the top images show listView with 439 width and the bottom images show listView with 285 width).

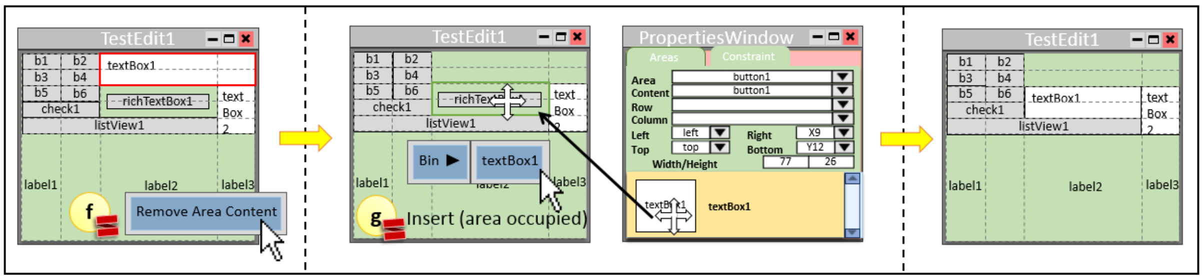


Figure 5-12 Edit operations and features in area mode part 2

Removing area content (Figure 5-12f)

To remove a particular component from the TestEdit1 window, either drag and drop the component outside of the TestEdit1 window, or by right-clicking on the component and selecting “Remove Area Content” menu item. This component then disappears from the TestEdit1 window and appear as a customized JLabel in the panel for storing bin components in the areas tab of the properties window (end result of the action in Figure 5-12 left is the textBox1 appearing in the bin in Figure 5-12 middle).

Reinserting a component from bin (Figure 5-12g)

To swap a component in bin with a component in TestEdit1, simply click on the component in the bin (e.g. textBox1 in Figure 5-12 middle) and drag the mouse to a location in the testing window (e.g. dragging to richTextBox1 in Figure 5-12 middle). Alternatively the same effect can be achieved by right-clicking on the richTextBox1 area and hovering over the “bin” menu and selecting the desired component in the bin to replace richTextBox1 (e.g. by selecting TextBox1 in Figure 5-12 middle). The resulting window is shown in Figure 5-12 right in which textBox1 has replaced richTextBox1 and richTextBox1 is now in the bin. A buggy implementation of the operation for inserting into an unoccupied area is also available and described in Section 6.3.4.

5.5.3. Features and Edit Operations in Constraints Mode

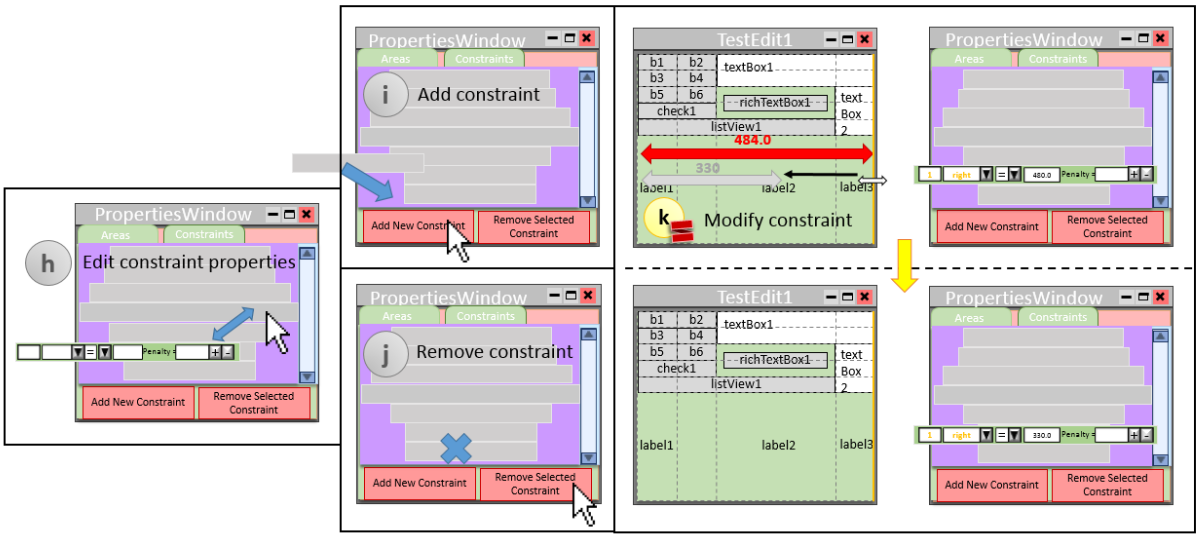


Figure 5-13 Edit operations and features in constraints mode part 1

Edit constraint properties (Figure 5-13h)

By clicking a particular constraint label, it is replaced by an editable form with text fields, combo boxes, and buttons and allow for the editing of the selected constraint (Figure 5-13h). Editable components include: the coefficient (text field), variable (combo box), operation (combo box containing =, <= or =>), rightSide (text field), penalty (text field), and two buttons to add or remove another summand to the constraint (see Figure 5-3 for an enlarged, labelled version of the constraint editing panel).

Add new constraint (Figure 5-13i)

Clicking the “Add New Constraint” button causes a new constraint to be added (with default values) to the list and the corresponding change of applying this constraint is seen in the testing window (Figure 5-13i).

Remove selected constraint (Figure 5-13j)

Clicking the “Removed Selected Constraint” while a constraint is selected from the list, causes the selected constraint to be removed from the list and the corresponding change of applying this constraint is seen in the testing window (Figure 5-13j).

Modifying a distance constraint (Figure 5-13k)

It is possible to drag and modify a distance constraint, which edits the rightSide of the constraint as well as edit the value in the text field. The constraint is being dragged and modified in Figure 5-13k top and after its release in Figure 5-13k bottom, the rightSide text field value is updated. However this operation is buggy and doesn’t change the GUI.

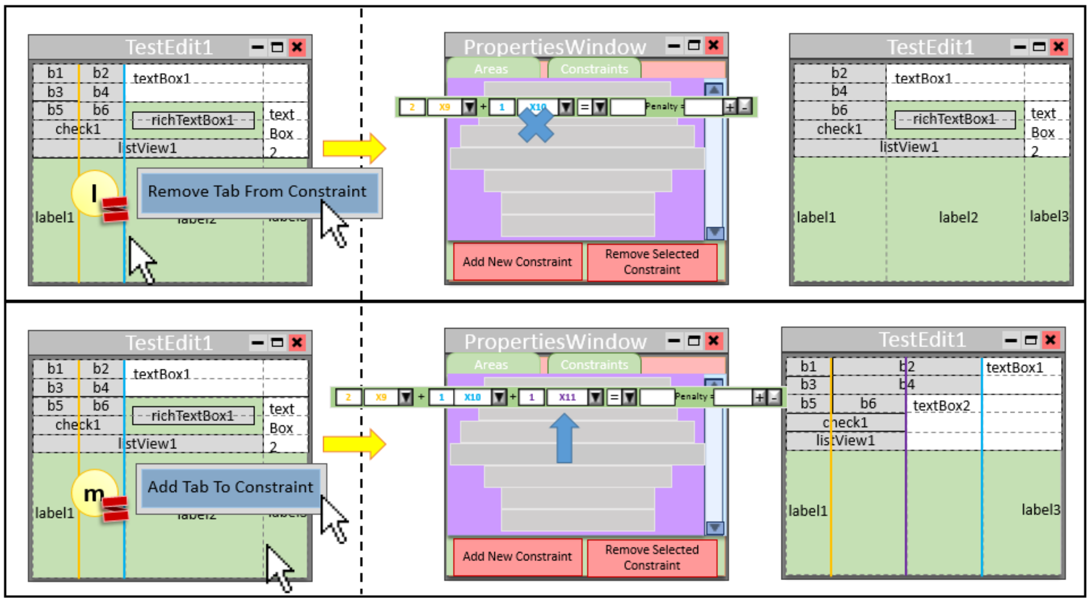


Figure 5-14 Edit operations and features in constraints mode part 2

These menu items change dynamically depending on the position of the right-click.

Remove tab from constraint (Figure 5-14l)

This is only available in the popup menu when the right click position is near one of the XTabs or YTabs in the currently selected constraint (in Figure 5-14l left, the constraint currently selected is 2.0 X9 - 1.0 X10 = 0.0). When the “Remove Tab from Constraint” menu item is clicked, the XTab near the right mouse-click (X10, the blue XTab) is removed from the currently selected constraint (which now becomes 2.0 X9 = 0.0) and the corresponding change of removing this constraint is be seen in the testing window (right-most image in Figure 5-14l right).

Add tab to constraint (Figure 5-14m)

This is only available in the popup menu when the right click position is near one the XTabs or YTabs not in the currently selected constraint (any of the grey dotted XTabs or YTabs, for example the XTab next to TextBox1 is selected in Figure 5-14m left). When the “Add Tab to Constraint” menu item is selected, the selected XTab or YTab is added to the currently selected constraint (ie another “+” and textbox with the default value 1.0 and combo box with the selected XTab or YTab). The selected constraint to be added is X11. In Figure 5-14m right, it appears in front of the “=” combo box. The corresponding change of adding this constraint is seen in the testing window (right-most image in Figure 5-14m bottom).

Chapter 6. Implementation

ALE described in chapter 5 is the state of the program at the start of this project before any changes were made (the old implementation). This chapter details the steps involved in converting the program to the new implementation at the end of this project. Figure 6-1 gives a general overview of the changes made to produce the new implementation. These changes relate to the four major objectives described in Chapter 1. 1) the conversion of the source code from Java to Scala, represented by the arrows in the figure (Section 6.1). 2) Refactoring of the source code to improve its quality (Section 6.2). This was mostly involved achieving better separation of concerns by creating edit operation classes. The relevant classes in the figure for this section are the deleted classes, the new edit operation classes, and the modified classes. 3) The classes highlighted with pink were used to modify existing or implement new functionality (Section 6.3). 4) Lastly, the new implementation of ALE was integrated into the IntelliJ IDEA through the creation of a prototype plugin (Section 6.3.1), as indicated by the grey box in the figure.

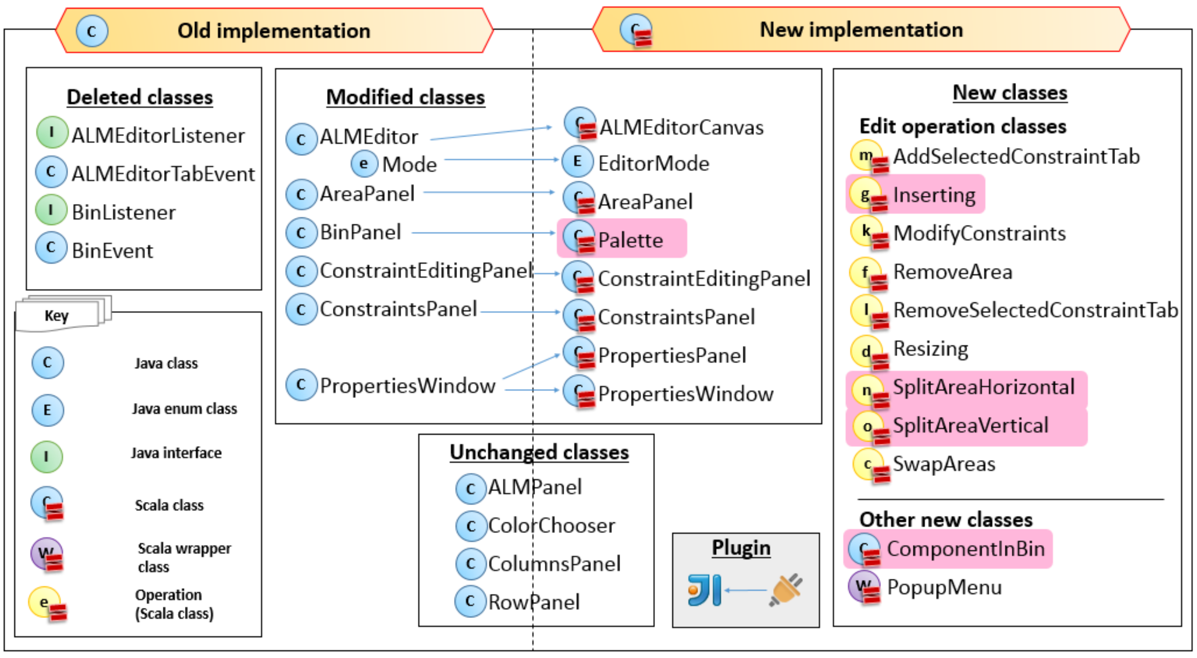


Figure 6-1 Overview of important ALE classes in the old vs the new implementation

6.1. Conversion of Java to Scala

The classes converted to Scala are listed in Figure 6-1, represented by the blue arrows going from a Java class to a Scala class. An automatic conversion tool in IntelliJ was used to partially convert each class into Scala (Section 6.1.1). The two main objectives were: 1) to convert the editable JComponents associated which perform the edit operations to Scala components to make use of Scala's more flexible event handling system (Section 6.1.3) 2) to make use of some of powerful Scala constructs, i.e. pattern matching (Section 6.1.2) and collection transformations (Section 6.1.4). During the process of converting the code to Scala, a custom wrapper class had to be created and a description is provided in Section 6.1.5. The following sections assume that the reader is familiar with basic Scala syntax and conventions. A general overview is presented in Appendix D for those unfamiliar with the language.

6.1.1. Initial Conversion from Java to Scala Classes

The "Convert Java file to Scala" offered by the Scala plugin supported by the IntelliJ IDE was used to automatically convert each Java class to Scala class as needed. The first step involved manually fixing the problematic code to conform to the Scala conventions so that each class would compile and run successfully. This meant that features not supported in Scala had to be removed and replaced with alternatives such as:

- the use of filtering statements instead of `continue`
- using the `breaks` class in `scala.util.control` which allows the use of the `break` keyword to exit an enclosing block marked with `breakable`
- importing `JavaConversions` to manage the conversion between Java and Scala collections so that the Java collections returned by a method or used in the code is automatically converted to a Scala buffer that can be used by Scala code
- Enum type is no longer supported for Scala so an enum class was created to store the mode constants
- Assigning a Java variable to a Scala variable requires use of the `_$eq` construct.
- Some of the methods were no longer working since Java components were expected but instead Scala components were being passed to it. Because Scala swing builds its library by "wrapping" the components in Java, the corresponding Java component could be obtained by using the `peer` method

After achieving successful compilation, there were still several runtime errors and setting issues preventing the testing of the application. These errors and their solutions are listed in Appendix C. After this initial conversion, the source code was in Scala syntax only but the code was still Java code in all other respects. The rest of this section describes the task of gradually converting the code to further utilise Scala's features.

6.1.2. Switch Statements in Java vs Pattern Matching in Scala

Case classes in Scala allow for pattern-matching on objects without a large amount of boilerplate. In Java the construct for a switch statement is as follows:

```
switch (selector) {//alternatives// //default at the end//}
```

while the construct for pattern matching in Scala is:

```
selector match {//alternatives//
                pattern1 => statements
                pattern2 => statements ...}
```

A sequence of `alternatives` (`pattern => statements`) is used for pattern matching, each starts with the keyword `case`. After the keyword is the `pattern` to be matched, followed by => and `statements` which are executed upon matching the `pattern`. The matching expression matches value `selector` against the patterns `pattern1`, `pattern2`, etc in the given order. There are many types of patterns in Scala. The constant pattern matches constants based on equality (as determined by ==), this is similar to Java's switch statements which also match on constants. Other types of patterns include the constructor pattern that matches on multiple arguments, the sequence pattern which matches on sequences or arrays, tuple patterns which match on tuples, typed patterns to test for type-testing. Therefore Scala allows for the matching of many more types than just the primitive types and String offered by Java.

```
  // Panels for tabbed pane in properties window, for selecting and using the
different editing modes
  var areaPanel: AreaPanel = new AreaPanel(//...//)
  var constraintsPanel: ConstraintsPanel = new ConstraintsPanel(//...//)

peer.add(new TabbedPane {
    pages += new Page("Areas", areaPanel)
    pages += new Page("Constraints", constraintsPanel)
    //because listenTo(selection) only calls the cases when a tab is pressed
    listenTo(this)
    reactions += {
      case e: ChangeEvent =>
        selection.page.title match {
    //gets the page from the selection, and shows whatever page whose title
matches the selection.page.title
          case "Areas" => aLMEditorCanvas.setMode(AreaEdit)
          case "Constraints" => aLMEditorCanvas.setMode(ConstraintEdit)
        }
        aLMEditorCanvas.repaint
    }
    peer.setVisible(true)
}.peer)
```

In the above code the page member from the `TabbedPane` class is used to add a new tab, passing its title and content panel as arguments. The matching of the title of each page to a particular case of type String is an example of constant pattern matching. `TabbedPane` contains the selection member which represents the current tab selection. The page title is retrieved from this member and then checked against each alternative to determine which is the currently selected page with respect to equality (==). Another example of pattern-matching in the code above is the `ChangeEvent` case, an example of applying case classes for pattern matching in Scala's event handling system which is further explored in Section 6.1.3.1 below. It also provides examples of the more powerful constructor pattern in Scala.

6.1.3. Event Handling in Java vs Scala

This subsection begins with a description of the event handling system in Java. Subsequently Scala event handling system is also described with particular emphasis on the constructs that help make it effective.

6.1.3.1. Event handing in Java

An example of Java event handling used in the code is:

```
areaBinPanel = new BinPanel();
areaBinPanel.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        // some code //
    }
    public void mouseReleased(MouseEvent e) {
        // some code //
    }

    public void mouseDragged(MouseEvent e) {
        // some code //
    }
});
```

In Java Swing, event handling occurs as follows (see Figure 6-2 for a diagram representation):

- Install an event listener object for the component by using the method (`.addEventXListener()`)

- The event listener object is passed as a parameter, which implements an interface with an `EventXHandler` method (this event listener object is usually created with an anonymous class)
- This event handler method receives an event object as a parameter, which can be used to infer information about the event that just occurred

When an event occurs:

- An event object is an object that is created when the user performs an action by interacting with the GUI
- The event object is sent to the specified target component
- Then the object notifies the event handler and the correct handler method is called with the object passed as an argument

6.1.3.2. Publisher/Subscriber System in Scala

An example of the same code from Section 6.1.3.1 written in Scala is (see Figure 6-2 for a diagram representation):

```
areaBinPanel = new BinPanel() {
  listenTo(this.mouse.clicks, this.mouse.moves)
  reactions += {
    case e: MouseDragged => // some code executed after mouse drag //
    case e: MousePressed => // some code executed after mouse press //
    case e: MouseReleased => // some code executed after mouse release //
  }
}
```

In Scala swing, an event is an object (or more specifically, an instance of a case class, explained in the next section) that is sent to a subscribing component. Each component may be a publisher and/or subscriber. A publisher (or "event source") publishes events while a subscriber (or "event listener") subscribes to a publisher and is notified when events are published. All classes with the `publisher` trait has the `publish` member which is a method used to notify all registered reactions All classes with the `reactor` trait has the `listenTo`, and `reactions` members. The `listenTo` member accepts publishers as arguments; the component containing the `listenTo` member becomes subscribed to the publishers passed as arguments (similar to installing listeners in Java). Each component inheriting from `reactor` has a collection of reactions: the += method is used to install/register reactions in addition to the standard ones. Similarly, the -= method can be used to uninstall/deregister a previously registered reaction.

`areaBinPanel` is instantiated as an instance of `BinPanel` (a Scala Panel class). Because `Panel` has the `reactor` trait, it can respond to events (it has the `listenTo` method). In this example, the panel is subscribed to mouse click and mouse movement events within itself. The handler refers to the block of code beginning just after reactions containing the case statements. Instead of the event object being passed to a particular `eventXHandling` method as in Java, all event handlers are contained in the handling block and a particular partial function system is used to handle the event (described in the next section).

Each component has a `Mouse` object which in turn contain three objects (also publishers) for dealing with different types of mouse events: clicked (`MouseClicked`, `MousePressed`, `MouseReleased`, `MouseEntered`, and `MouseExited` events.), moves (`MouseMoved` and `MouseDragged` events), and wheel publishers(`MouseWheelMoved` events). Similarly it has a `Keys` object for dealing with key events. In the example above, calling `listenTo` on only the mouse objects means the handler is only called in response to mouse clicks and moves. If `listenTo(this)` is called instead, then the handler would react to every type of event fired from the bin panel.

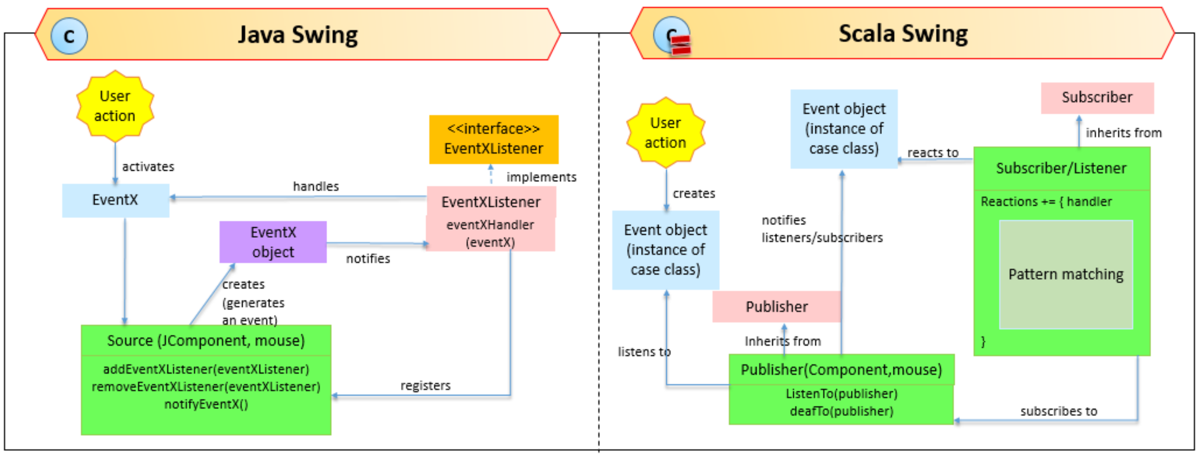


Figure 6-2 Diagram showing the difference between Java and Scala event handling

6.1.3.1. Partial Functions, Pattern Matching and Case Classes in Scala

A partial function can only operate on certain values of the arguments and is undefined for other arguments. The event handlers are partial functions that pattern match on events. Just like in Java, there are many different subtypes of `Event` for responding to different types of events; in Scala they are defined in the `Scala.swing.event` package. Unlike Java though, event handling in Scala can make use of case classes, which are designed to support pattern matching succinctly. An instance of the case class is created when user interaction initiates an event. For example when a mouse is pressed, the following case class is created:

```
  case class MousePressed(source: Component, point: java.awt.Point, modifi-
ers: Modifiers, clicks: Int, triggersPopup: Boolean) (peer:
java.awt.event.MouseEvent)
```

The event object is used for pattern matching in the handler (the reactions block). In the code example provided in Section 6.1.3.2 above:

- this partial function operates on events, but would only be defined if the argument was of `MouseDragged`, `MousePressed`, or `MouseReleased` type
- Only mouse events is be passed to the handler since `listenTo` method is only registered on mouse clicks and movements
- if the user clicked on the bin panel, then the `MousePressed` and `MouseReleased` event objects are created (instances of the `MousePressed` and `MouseReleased` case classes)
- These patterns check for the type of event object given to it. For example `case e: MousePressed` matches anything that is an instance of the `MousePressed` case class. As seen here, it is possible to match on more than one type of event in a single handler by using multiple event types.
- If the pattern is matched, the handler actions that correspond to this event in the code would be executed (the code after =>)

Scala's pattern matching expressions can use constructors as patterns, which can be very useful for GUI applications. The example below uses patterns of the form `Constructor(var1,var2)` where `Constructor` is a case class constructor and `var1` and `var2` are variables. In a handler, the object is first checked if it is an instance of the named case class, then the constructor parameters are checked. The constructor parameters offer a way to provide more specific conditions in a pattern.

In the constructor pattern below the variable `checkComponent` is passed as the `source:Component` argument. The underscore signifies the wildcard, so those arguments are ignored in the pattern. Only a `MousePressed` event object created by clicking on that specified component produces a match, clicking on anything else does not produce a match.

```
case s: MousePressed('checkComponent',_,_,_,_)
```

More arguments can be passed for "deeper" checking, such as only matching when the specified component is clicked exactly twice:

```
case s: MousePressed('checkComponent',_,_,2,_)
```

In conclusion, event handling in Scala seems to be more powerful and concise due to the use of concepts mentioned above: publisher/subscriber, partial functions, pattern matching in conjunction with case classes.

6.1.4. Collections Transformations in Scala

Scala's collection methods allow for more concise syntax compared when transforming collections. The example code below comes from the `PropertiesWindow` class and illustrate that Scala does not require the use of loops or explicit iterations to perform collection transformations. In the first example, the array list of constraints in the layout specification is transformed into a list minus the constraints without an owner. The underscore is a placeholder for one or more parameters, so `_.Owner==null` is very short notation for a function that checks whether the constraint's owner is null or not. In the Java version, a new array list created, a for-loop goes through each constraint and adds only those whose Owner is not null. This process that takes up 6 lines of code in the Java version is achieved in one line in Scala.

```
userConstraints = le.getLayoutSpec.getConstraints filter (_.Owner == null)
```

The second example illustrates the use of map on a collection. Here the array list `constraintLabels` contains a list of type `scala.swing.Label` but it needs to be transformed into an array list containing the names of its corresponding `javax.swing.JLabel`. Instead of iterating through a for-loop and changing each item one at a time, the map transformation is applied:

```
constraintLabels map (_.name))
```

6.1.5. Making a Custom Wrapper Class in Scala

Due to the enormous number of JComponents available, not all wrapper classes are provided for their Scala counterparts. In this subsection we describe the steps involved in making a custom wrapper class (Section 6.1.5.3). We begin by introducing concepts relevant to its creation, namely: Scala traits (Section 6.1.5.1) and the Scala wrapper system (Section 6.1.5.2).

6.1.5.1. Scala traits

The trait construct is unique to Scala inspired by Java's interfaces and Ruby's mixins. In addition to abstract members, they can also have full method definitions (concrete value members) and allow for the inheriting classes to possess multiple traits. The notorious diamond problem produced by multiple inheritance is avoided in traits by the way they are treated in an inheriting class. Traits are not true superclasses but are actually mixins as in the concrete value members are treated as though they belong in the inheriting class. Therefore there is no inheritance path ambiguity, thus avoiding the diamond problem. If two traits have the same method (name, parameter, return type), the inheriting class will use the method from the "dominant" trait. For example, for the following class, `Trait1` is dominant because it was declared first.

```
class TestClass (number:Int) extends Trait1 with Trait2
```

In this way, many traits can be mixed into the class by using the `extends` keyword for the superclass or predominant trait and any number of traits after it using the `with` keyword.

6.1.5.2. Scala Swing Hierarchy and Wrapper System

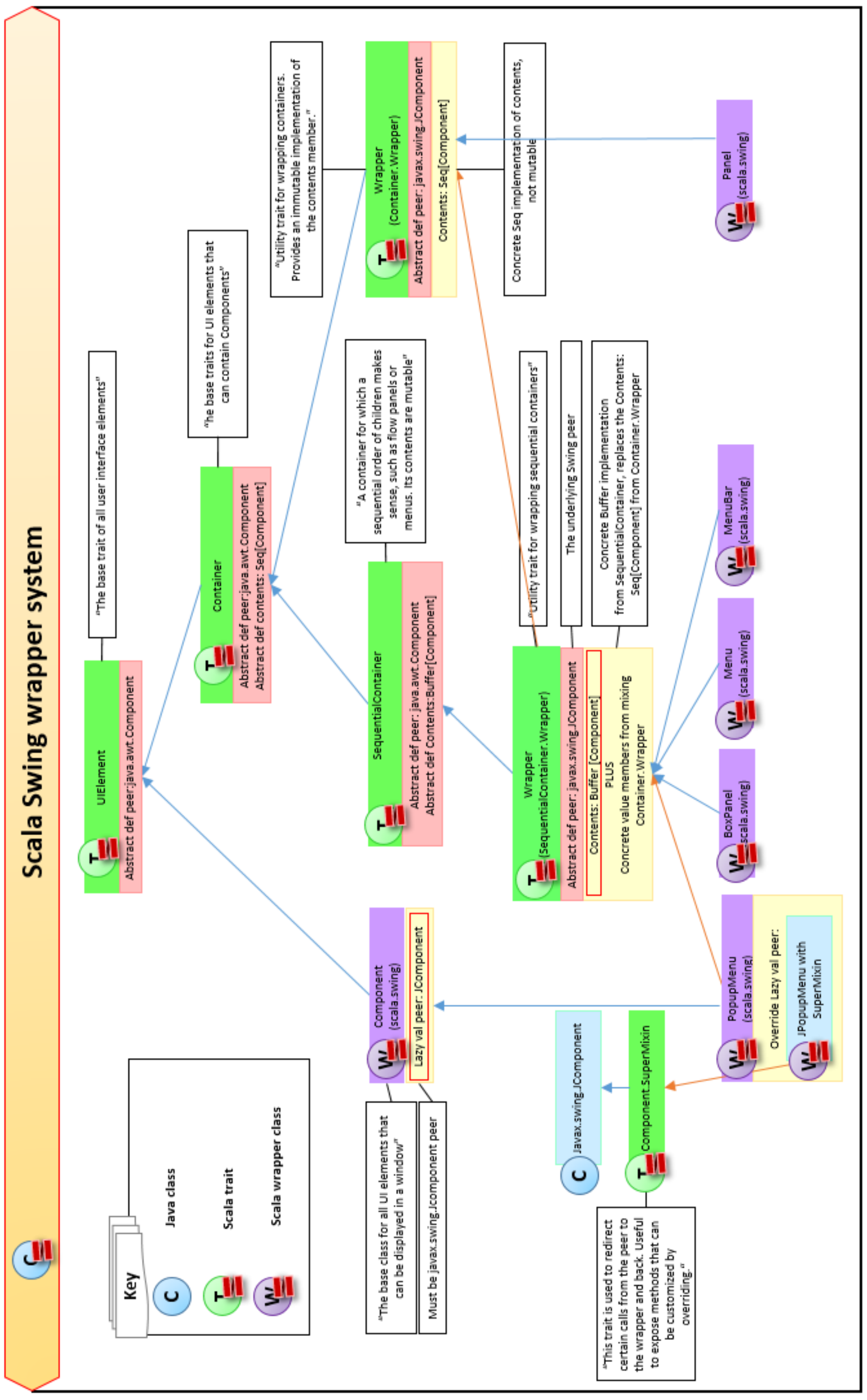

Scala Swing wrapper system
Key
C Java class
T Scala trait
W Scala wrapper class
UIElement
Abstract def peer:java.awt.Component
"The base trait of all user interface elements"
Container
Abstract def peer:java.awt.Component
Abstract def contents: Seq[Component]
"he base traits for UI elements that can contain Components"
Wrapper (Container.Wrapper)
Abstract def peer: javax.swing.JComponent
Contents: Seq[Component]
"Utility trait for wrapping containers. Provides an immutable implementation of the contents member."
Concrete Seq implementation of contents, not mutable
SequentialContainer
Abstract def peer: java.awt.Component
Abstract def Contents:Buffer[Component]
"A container for which a sequential order of children makes sense, such as flow panels or menus. Its contents are mutable"
Wrapper (SequentialContainer.Wrapper)
Abstract def peer: javax.swing.JComponent
Contents: Buffer [Component]
PLUS
Concrete value members from mixing Container.Wrapper
"Utility trait for wrapping sequential containers"
The underlying Swing peer
Concrete Buffer implementation from SequentialContainer, replaces the Contents: Seq[Component] from Container.Wrapper
Panel (scala.swing)
MenuBar (scala.swing)
Menu (scala.swing)
BoxPanel (scala.swing)
PopupMenu (scala.swing)
Override Lazy val peer:
JPopupMenu with SuperMixin
Component (scala.swing)
Lazy val peer: JComponent
"The base class for all UI elements that can be displayed in a window"
Must be javax.swing.Jcomponent peer
javax.swing.JComponent
Component.SuperMixin
"This trait is used to redirect certain calls from the peer to the wrapper and back. Useful to expose methods that can be customized by overriding."

Figure 6-3 The Scala Swing wrapper system

This figure follows the colour-coding system of Figure 3-1. The scala.swing classes: those with an abstract peer members in a green box while those with a concrete peer member is in a purple box. The abstract members are listed in the red box and concrete members are listed in the yellow box below the scala.swing element. Classes from the javax.swing package are surrounded by a blue box. Comments are in white boxes pointing to the element or member they are describing. The comments in quotation marks are directly copied from the Scala API

Figure 6-3 gives an outline of a section of the Scala Swing hierarchy and shows where the custom-made `PopupMenu` is situated within this structure. `UIElement`, `Container` and `SequentialContainer` are all traits which define members for their specific purposes. `UIElement` is the base class and defines concrete members needed by all GUI components such as `background`, `size`, `repaint`, and `reactions`. The `Container` trait has the abstract member `contents` (immutable sequence of child components) which is implemented by `Container.Wrapper`, a utility trait for wrapping `Container`. The `SequentialContainer` refines the abstract member in `Container` to become a mutable buffer of child components, which allows for addition and removal in a specific order. SequentialContainer.Wrapper extends SequentialContainer, implements its contents member, and thus has a mutable buffer to store its child components. It also inherits from the Container.Wrapper trait and mixes in some concrete value members it.

Components which require a sequential ordering of its child components (e.g. BoxPanel, Menu, and MenuBar) extends the SequentialContainer.Wrapper trait and each of them can make use of the concrete contents member. In contrast, child components of the Panel class is stored in a sequence and is immutable, because sequential ordering is not necessary. In summary, a programmer can make use of the various properties provided by these traits to make a customized component. For example the component should inherit SequentialContainer.Wrapper if sequential ordering of its child components makes sense but inhert Container.Wrapper if the list containing its child components should not be modified.

6.1.5.3. Creating a PopupMenu Wrapper class

A PopupMenu wrapper class was created for this project as no wrapper exists for JPopupMenu. Two important inherited members are the contents mutable buffer (from SequentialContainer.Wrapper, already mentioned above) and the peer member (from Component) (highlighted with red boxes in Figure 6-3). Notice that the JPopupMenu mixes in the Supermixin trait, which overrides methods within javax.swing Component such as paintComponent, paintBorder and paintChildren; it is used to redirect certain calls from the peer to the wrapper component.

These are the main steps required for defining a custom wrapper class (with the PopupMenu as an example). 1) It must extend scala.swing.Component, so it can be classified as a UI Component and make use of the related functionalities. 2) It needs to inherit traits that define concrete methods needed for the class. The PopupMenu extends Component with SequentialContainer.Wrapper, so it has the ability to have sequential ordering of child components and to add them using the contents.+= construct. 3) Most importantly, the peer member has to be overridden with a class from the javax.swing package with SuperMixin, in this case it is javax.swing.JPopupMenu 4) The last step is to create peer methods, in this example it is JPopupMenu.show which makes the right click menu visible in the position of the click. The code for the wrapper class is shown below:

```
/**Custom wrapper for JPopupMenu since PopupMenu no longer exists in scala*/
class PopupMenu extends Component with SequentialContainer.Wrapper {
  override lazy val peer: JPopupMenu = new JPopupMenu with SuperMixin

  // create peer methods here
  def show(component: Component, xPos: Double, yPos: Double): Unit =
peer.show(component.peer, xPos.asInstanceOf[Int], yPos.asInstanceOf[Int])
}
```

6.2. Refactoring ALE

6.2.1. General Refactorings

Several common refactoring techniques mentioned in the refactoring catalogue defined by Fowler et al [14] were applied in this study and is referenced in this subsection in square brackets, e.g. [move class].

Before major changes to the ALE class system was made, some smaller changes were made in the classes to improve inefficient, complex, or hard-to-understand code. The update method was a long method extracted to become areaUpdate and constranintUpdate methods containing code related to area and constraint update, respectively [extract method]. The old implementation had one popup menu for all menu items in editing mode and checked the mode to determine which one to display. Two separate popup menus were created in the new implementation, one for each mode [extract field]. Some simple methods were only called once, therefore it the method was removed and its code integrated into the correct position in the code (Remove middle man). The `PropertiesWindow` class was moved from `alm` package into `alm.editor` package, since it is a part of the editor [move class]. Other refactoring such as [renaming methods/ fields/classes] (e.g. `BinPanel` renamed as `Palette`, `ALMEditor` renamed as `ALMEditorCanvas` to better reflect their true purpose), [encapsulate field/method] was also performed.

One example of refactoring to improve the efficiency of a method is the `getAreaYTabSelected` method. It finds whether the mouse clicked on the top YTab or the bottom YTab of the selected area, within a given tolerance (`TOL`). It takes in a parameter representing the currently selected area (`selectedArea`) and a point representing the mouse click position (`point`). The unrefactored method called the `getYTabSelected` method, which looped through all X/YTabs in the layout specification and returned the first found YTab with the position corresponding to the mouse press position (within a certain distance given by the tolerance). Then, if the clicked YTab is either the top or the bottom border of the area, it is returned. This method is called every time the mouse is pressed and is clearly inefficient since its worst case complexity is O(n) where n is the number of X/YTabs in the layout specification, i.e. every time the mouse does not click on a YTab.

```
  def getAreaYTabSelected (selectedArea: Area, point: Point): Variable = {
    // Call a minimum search algorithm for the nearest YTab
    val selectedYTab: YTab = getYTabSelected(point).asInstanceOf[YTab]
    // Check that the nearest YTab is either the top or bottom border of the 
currently selected area
    if (selectedArea.getTop == selectedYTab || selectedArea.getBottom == 
selectedYTab) return selectedYTab
    else return null
  }
```

The refactored version is shown below. It minimizes the steps for the search by only considering the top YTab and the bottom YTab for the `selectedArea`. It simply checks whether the mouse press point corresponds to the top YTab or the bottom YTab position within a certain distance given by the tolerance. This algorithm always requires only a constant number of steps and this should increase the performance of ALE, especially in a larger example with more variables in the layout specification. Figure 6-4 is a pictorial representation of the position of each variable used in the method, for example `selectedArea.getTop.getValue + TOL` indicates a y-coordinate a certain distance (`TOL`) below the top border of the area (`selectedArea.getTop.getValue`). The mouse click must be within the region highlighted in blue to select the top border YTab.

```
  def getAreaYTabSelected (selectedArea: Area, point: Point): Variable = {
    // test if it is the top tabstop
    if (point.y >= selectedArea.getTop.getValue && point.y <= 
selectedArea.getTop.getValue + TOL) return selectedArea.getTop
    // test if it is the bottom tabstop
    if (point.y >= selectedArea.getBottom.getValue - TOL && point.y <= 
selectedArea.getBottom.getValue) return selectedArea.getBottom
    // otherwise: no tabstop
    return null
  }
```

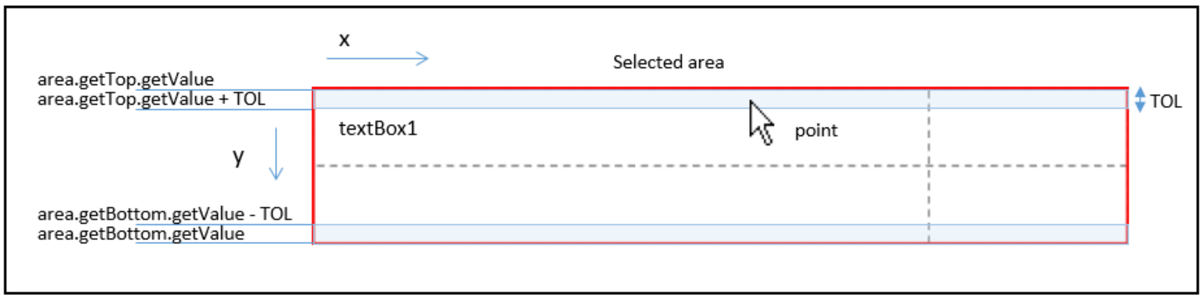


Figure 6-4 The important variables in the refactored getAreaYTabClicked method

6.2.2. Separation of Concerns

Separation of Concerns (SoC) is a design principle for separating a program into distinct parts such that each part addresses a different concern. A concern has been defined as a part of a program “relevant to a particular concept, goal, or purpose” [52]. Since ALE is a GUI application, it was refactored so that each part (a GUI top-level container class) contains all its child components (Section 6.2.2.1); and also contains all the necessary methods and fields to perform well-bounded functionalities (Section 6.2.2.2). The advantages of such a structure is further explored in (Section 7.1.1.1).

6.2.2.1. Separation of GUI Components

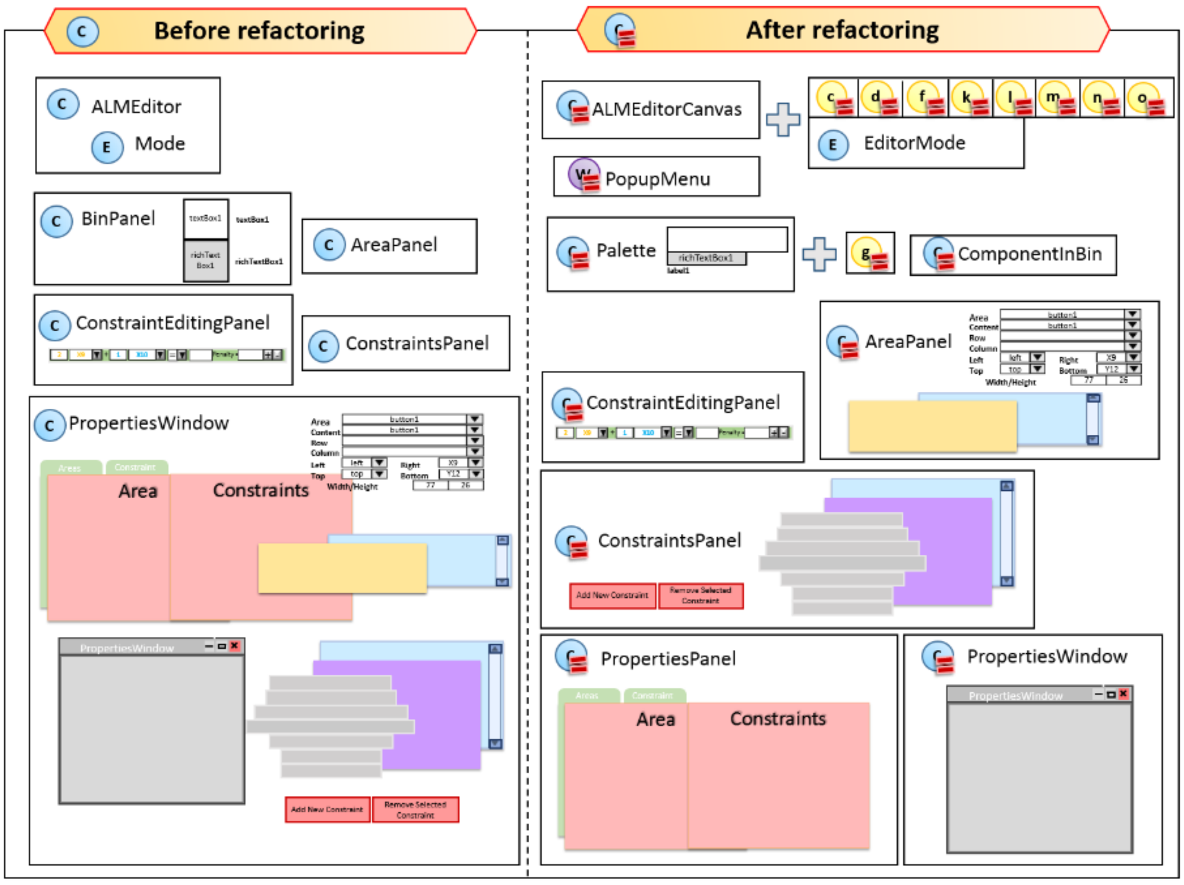


Figure 6-5 Separation of components before and after refactoring

Note that for completeness, this figure also shows the modified and newly implemented features described in Section 6.3. Edit operations n and o, plus the changed appearance of the components in the palette.

Figure 6-5 shows the distribution of components across the classes in the `alm.editor` package (where each class is encased with a black border) before and after refactoring. This figure follows the conventions used in Section 5.5 in that the features and operations are indexed by letters contained in grey and yellow circles, respectively. `PropertiesWindow` was split it into two classes: `PropertiesWindow`, which contained the code that a Frame (Window) needs to use (i.e. the menu bar), and `PropertiesPanel`, which contains the remaining code. Most of the components in the `PropertiesWindow` class was then moved into other classes. After refactoring, the `PropertiesPanel` class contains the `tabbedPane` and points to `AreaPanel` and `ConstraintsPanel`. The area panel’s child components (combo boxes, labels, `Palette`, `areaScrollPane`) were moved to the `AreaPanel` class; it also points to the `Palette` class. The constraints panel’s child components (Buttons, Labels, `ConstraintsListPanel`, and `constraintsScrollPanel`) were also moved into the `ConstraintsPanel` class; it also points to the `ConstraintsEditingPanel` class.

6.2.2.2. Separation of Functionalities

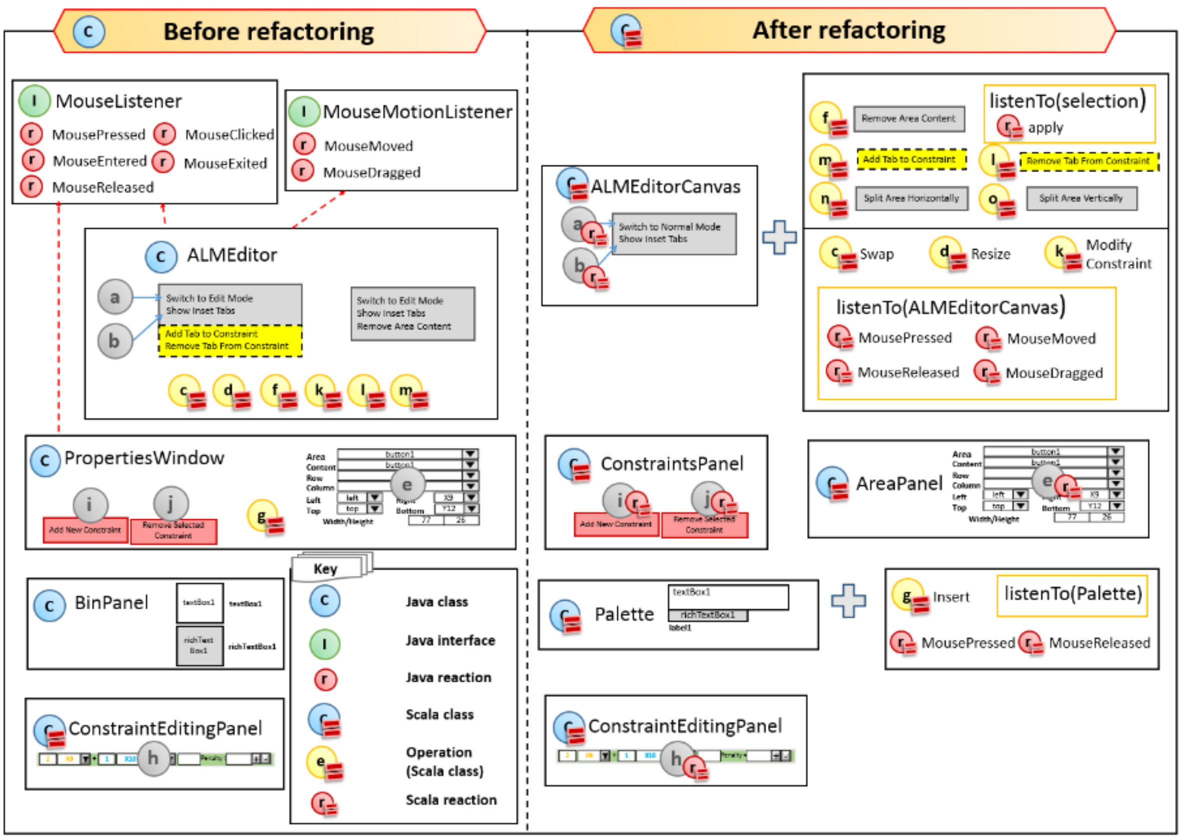


Figure 6-6 Separation of features/operations before and after refactoring

Note that for completeness, this figure also shows the modified and newly implemented features described in Section 6.3. Edit operations n and o, plus the changed appearance of the components in the palette.

Figure 6-6 shows the distribution of functionality in each class (where each class is encased in a black box) before and after refactoring. The components are indexed with the functionality they perform. This figure follows the conventions used in Section 5.5 in that the features and edit operations are indexed by letters contained in grey and yellow circles, respectively. Most of the functionality contained in the `ALMEditor` class was factored out into edit operation classes (as Figure 6-6 shows that the code becomes `ALMEditorCanvas` plus many operation classes). Similarly, the insert edit operation was moved out from `PropertiesWindow` but is now associated with the `Palette`, as it needs to listen to events fired from the palette panel. The methods and fields performing functionality in properties window was further separated into `ConstraintsPanel` (features i and j) and `AreaPanel` (feature e) because they are associated with the components now contained within these classes.

Before refactoring, `ALMEditorCanvas` extended `MouseListener` and `MouseMotionListener`; `PropertiesWindow` also extended `MouseListener` and overrode the `eventXHandler` methods for reacting to user action. The code for the edit operations was spread out across the `MousePressed/Released/Moved/Dragged` Scala event handling cases in the `ALMEditorCanvas` class. After refactoring, the code belonging to each operation is in its own edit operation class, therefore achieving separation of functionality. The edit operation classes which provide functionality for `ALMEditorCanvas` can be separated into two groups. The first group (classes for operations f, l, m, n, o) all contain the code corresponding to an action executed after clicking a menu item. As such, they all extend the `Action` abstract class and implement the `apply` method and listen to the selection object provided by the `Combo box` class. The second group (classes for operations c,d,k) listens to the `ALMEditorCanvas` and simply installs more reactions for the canvas (by using += method to add more event handling capability). They do not extend the `Action` class but instead contain code for their specific editing operation within the `MousePressed/Released/Moved/Dragged` cases.

Below is the example code for the `RemoveArea` (operation f) operation class showing the general structure of the first group of operations:

```
class RemoveArea(canvas: ALMEditorCanvas) extends Action("Remove area
content") {
   // if this menuItem is pressed, call apply
  canvas.areasPopupMenu.contents += new MenuItem(this)
  // install reactions for direct manipulation invocation of this operation
  canvas.reactions += {
        case e: MousePressed if (canvas.editorMode == EditorMode.AreaEdit)=>
        // select area under mouse
        case e: MouseReleased if (canvas.editorMode == EditorMode.AreaEdit)=>
        // If the mouse is released outside of the main window
        // then call apply
  }

  /**
   * Removes the area, adds the content to the bin and refreshes the GUI.
   */
  def apply {
    // some code //
  }
}
```

- It extends `Action` which takes a string parameter as the name
- The popup menu for the area mode (`areasPopupMenu`) is accessed from `ALMEditorCanvas`. A new menu item is created by passing the `RemoveArea` class itself (an `Action` class) as an argument. It is then added to the popup menu.
- Reactions related to detecting the removal of components/areas are installed for `ALMEditorCanvas`
- The `apply` method is implemented to contain code which removes the area/component

Below is the example code for `SwapAreas` (operation c) operation class showing the general structure of the second group of operations:

```
class SwapAreas(canvas: ALMEditorCanvas){
  //install reactions for the canvas
  canvas.reactions+={
    case e: MousePressed if (canvas.editorMode == EditorMode.AreaEdit)=>
      // the mouse cursor is changed to MOVE_CURSOR, the area underneath the
mouse cursor is found and assigned to selectedArea
    case e: MouseReleased if(canvas.editorMode == EditorMode.AreaEdit &&
canvas.selectedArea != null) =>
      // the area under the initial mouse click (selectedArea)
      // and the area the mouse is currently over which is also the area at
the position of mouse release (mouseOverArea)
    case e: MouseDragged if (canvas.editorMode == EditorMode.AreaEdit) =>
      // the area underneath the mouse cursor when being dragged is found and
assigned to mouseOverArea
    case e: MouseMoved if (canvas.editorMode == EditorMode.AreaEdit) =>
      // the area under the mouse while moving is detected and assigned to
mouseOverArea
    }
}
```

- It installs reactions for the canvas and then specifies the event cases
- Together, the code within the `MousePressed/Released/Dragged/Moved` cases contain all the necessary steps for swapping an area

- The area under the initial mouse press is assigned to `selectedArea`. The area under the mouse as it moves is detected by `MouseDragged/Moved` and assigned to `mouseOverArea`. When mouse is released, the two areas swap.

Then, the edit operation class is instantiated in `ALMEditorCanvas`:

```
// Installing the removeArea editing operation in ALMEditorCanvas
val removeArea = new operations.RemoveArea(this)
// Installing the swap areas editing operation in ALMEditorCanvas
val swapAreas = new operations.SwapAreas(this)
```

6.2.3. Refactoring ALMEditorListener and BinListener

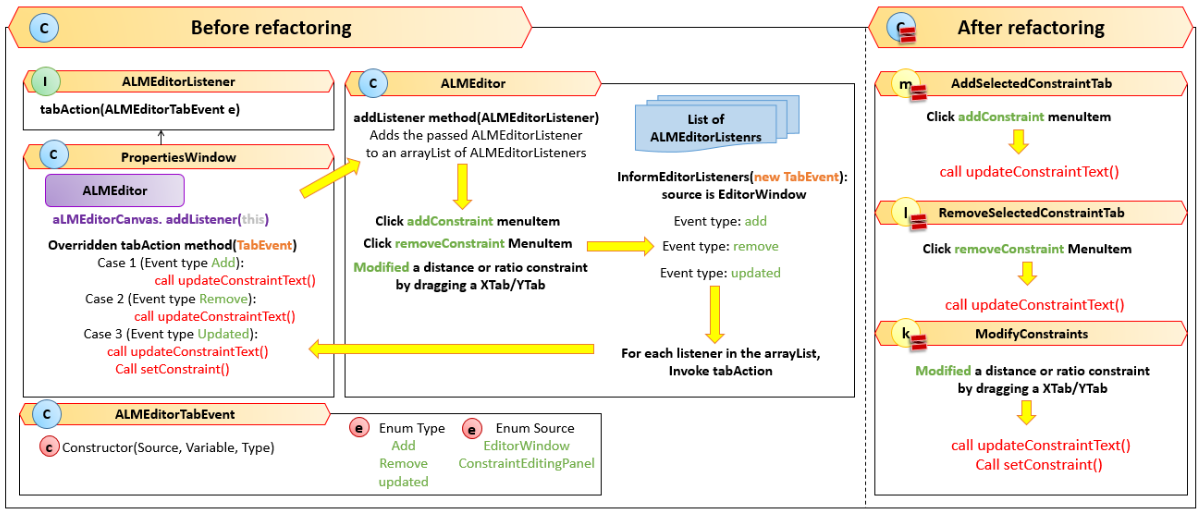

Before refactoring
After refactoring
ALMEditorListener
tabAction(ALMEditorTabEvent e)
PropertiesWindow
ALMEditor
aLMEditorCanvas. addListener(this)
Overridden tabAction method(TabEvent)
Case 1 (Event type Add):
call updateConstraintText()
Case 2 (Event type Remove):
call updateConstraintText()
Case 3 (Event type Updated):
call updateConstraintText()
Call setConstraint()
ALMEditor
addListener method(ALMEditorListener)
Adds the passed ALMEditorListener
to an arrayList of ALMEditorListeners
List of ALMEditorListenrs
InformEditorListeners(new TabEvent):
source is EditorWindow
Click addConstraint menuItem
Click removeConstraint MenuItem
Modified a distance or ratio constraint by dragging a XTab/YTab
Event type: add
Event type: remove
Event type: updated
For each listener in the arrayList, Invoke tabAction
ALMEditorTabEvent
Constructor(Source, Variable, Type)
Enum Type
Add
Remove
updated
Enum Source
EditorWindow
ConstraintEditingPanel
AddSelectedConstraintTab
Click addConstraint menuItem
call updateConstraintText()
RemoveSelectedConstraintTab
Click removeConstraint MenuItem
call updateConstraintText()
ModifyConstraints
Modified a distance or ratio constraint by dragging a XTab/YTab
call updateConstraintText()
Call setConstraint()

Figure 6-7 Refactoring the ALMEditorListener system

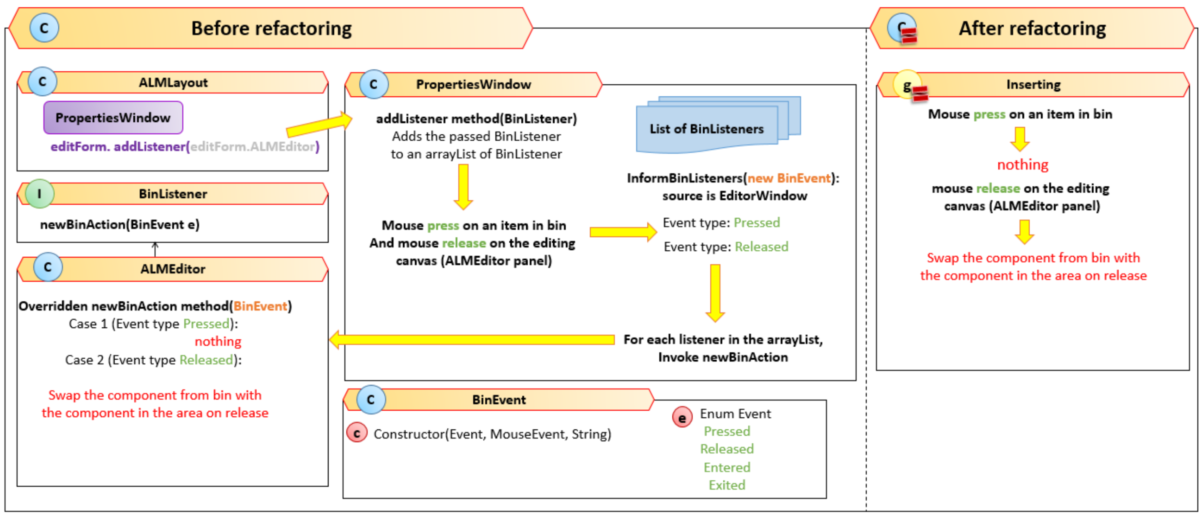

Before refactoring
After refactoring
ALMLayout
PropertiesWindow
editForm. addListener(editForm.ALMEditor)
BinListener
newBinAction(BinEvent e)
ALMEditor
Overridden newBinAction method(BinEvent)
Case 1 (Event type Pressed):
nothing
Case 2 (Event type Released):
Swap the component from bin with the component in the area on release
PropertiesWindow
addListener method(BinListener)
Adds the passed BinListener
to an arrayList of BinListener
List of BinListeners
InformBinListeners(new BinEvent):
source is EditorWindow
Mouse press on an item in bin
And mouse release on the editing canvas (ALMEditor panel)
Event type: Pressed
Event type: Released
For each listener in the arrayList, Invoke newBinAction
BinEvent
Constructor(Event, MouseEvent, String)
Enum Event
Pressed
Released
Entered
Exited
Inserting
Mouse press on an item in bin
nothing
mouse release on the editing canvas (ALMEditor panel)
Swap the component from bin with the component in the area on release

Figure 6-8 Refactoring the BinListener system

Before refactoring, two custom Listener classes and two custom event classes were used to respond to the adding/removing/updating of constraints (`ALMEditorListener` and `ALMTabEvent`) and to pressing/releasing the mouse on the bin panel (`BinListener` and `BinEvent`). After refactoring, the same functionality is achieved by removing the custom listener and event classes and integrating their functionality into the correct classes (Inline class).

ALMEditorListener/ALMEditorTabEvent (Figure 6-7)**:** `PropertiesWindow` implements the `ALMEditorListener` interface. Its instance in `ALMEditorCanvas` is added to an array list of

ALMEditorListeners by using the addListener method defined in ALMEditorCanvas. Whenever addConstraint or removeConstraint menu items are clicked, or when a constraint is modified by dragging an X/YTab, informEditorListeners is called and is passed a new ALMEditorTabEvent as a parameter. It passes the event type add, remove or updated for add constraint, remove constraint, or the modification action, respectively. Next, the tabAction is called for each listener in the ALMEditorListeners array list (note there is only one). Finally, the tabAction method defined within PropertiesWindow will perform a particular action depending on each event type. After refactoring, the necessary code is simply inlined into the edit operation classes designed for adding/removing/modifying a constraint.

BinListener/BinEvent (Figure 6-8)**:** Similar to the above situation before refactoring, ALMEditor implemented the BinListener and PropertiesWindow contained the list of listeners, the addListener and informListeners methods. An instance of ALMEditor was added to the list of BinListeners in the ALMLayout class. Again, in the new implementation everything is simplified by inlining the code immediately (within the Inserting operation class, in MousePressed and MouseReleased reaction cases).

6.2.4. Conclusion

This section has addressed the research question: "How can a complex GUI application (such as ALE) be refactored?" We have shown that general refactoring techniques can be applied to a complex GUI application like ALE. We have also demonstrated that functionality from a component required to handle many events (like ALMEditorCanvas) can be extracted into smaller classes. Our experiences finds Scala's reaction handlers useful for breaking reaction handling code into separate "concerns" each dedicated to a single functionality. Finally, we showed that removal of custom listener systems in a complex GUI application like ALE can be achieved by studying the call hierarchy and inlining the code.

6.3. Modifying Existing and Implementing New Functionality

6.3.1. Split Area Horizontally/Vertically

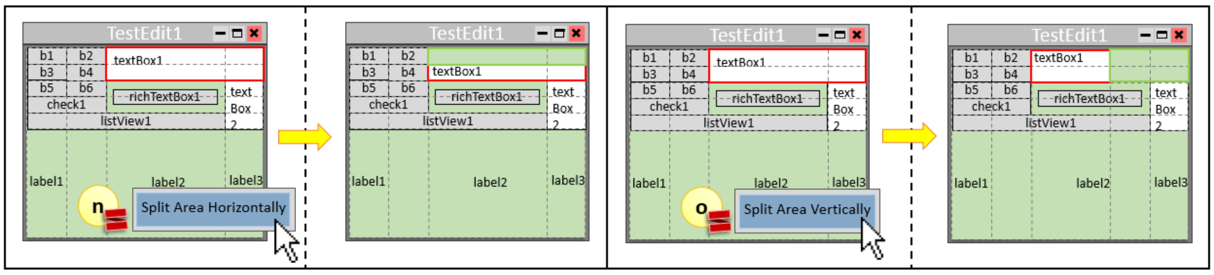


Figure 6-9 Newly implemented edit operations usable in both modes

Split area horizontally (Figure 6-9n)

The textBox1 in figure Figure 6-9n left is the old area. A new area is created (top half of the old area, highlighted in green in Figure 6-9n right and the old area is shrunken to become the current selected area (bottom half of the old area, highlighted in red in Figure 6-9n right). This was implemented by these steps:

- Create a new YTab
- set the value of the YTab halfway between the top and bottom borders of the old area
- Make a new area half the height of the old area and set the content to be null, it will be the top half of the old area
- Set the preferred size of the new area to be the same width as the old area but half the height of the old area
- Change the selected area's top tab to be the new tab created (thus the selected area is now on the bottom half of the old area)

Split area vertically ((Figure 6-9o)

The textBox1 in Figure 6-9o left is the old area. A new area is created (right half of the old area, highlighted in green in Figure 6-9o right and the old area is shrunken to become the current selected area (left half of the old area, highlighted in red in Figure 6-9o right). This was implemented by these steps:

- Create a new XTab
- set the value of the XTab halfway between the left and right borders of the old area
- Make a new area half the width of the old area and set the content to be null, it will be the right half of the old area
- Set the preferred size of the new area to be the same height as the old area but half the width of the old area
- Change the selected area's right tab to be the new tab created (thus the selected area is now on the left half of the old area)

6.3.2. Changes to the Appearance of Components Added to the Bin

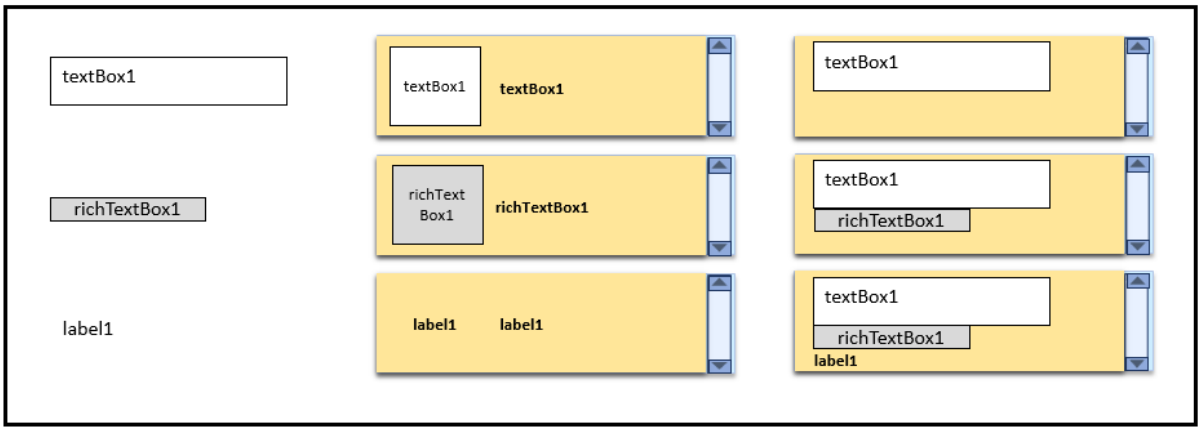


Figure 6-10 Contrasting the appearance of text boxes, buttons and labels in the old implementation versus the new implementation

In the old implementation, a custom JLabel was made for every component. A square was made containing the component name in the centre, with the background colour as background colour of the component. Another JLabel was placed next to the custom JLabel and everything was arranged using gridBagLayout layout manager (Figure 6-10 left panels). In the new implementation, buffered images are used so the direct images of the actual component (capturing the actual size in the layout) can be seen in the bin (Figure 6-10 right panels).

6.3.3. Changes to Insertion into an Already Occupied Area

Figure 6-11 shows the differences between the old and new implementation for inserting into an already occupied area. In the old implementation, an array list was used to store the JComponents in the bin while two hash maps were used to link the JComponent to its insets or horizontal/alignments. Therefore when removing a component/area, the corresponding entry had to be removed from the array list and hash maps and when inserting a component/area the corresponding entry had to be added to them. In the new implementation, the properties (content, insets, horizontal/vertical alignments) are set up in the class `ComponentInBin` (which contains all information belonging to a particular bin item). The steps involved with swapping the `selectedArea` (area in edit canvas in the mouse released position) with the `binItem` (item selected in the bin) is therefore much simpler in the new implementation as everything can be accessed from the `ComponentInBin` class. Furthermore, the new implementation no longer uses `BinListener` and `BinEvent`, and the code has been separated into edit operation classes; hence the new implementation is easier to understand.

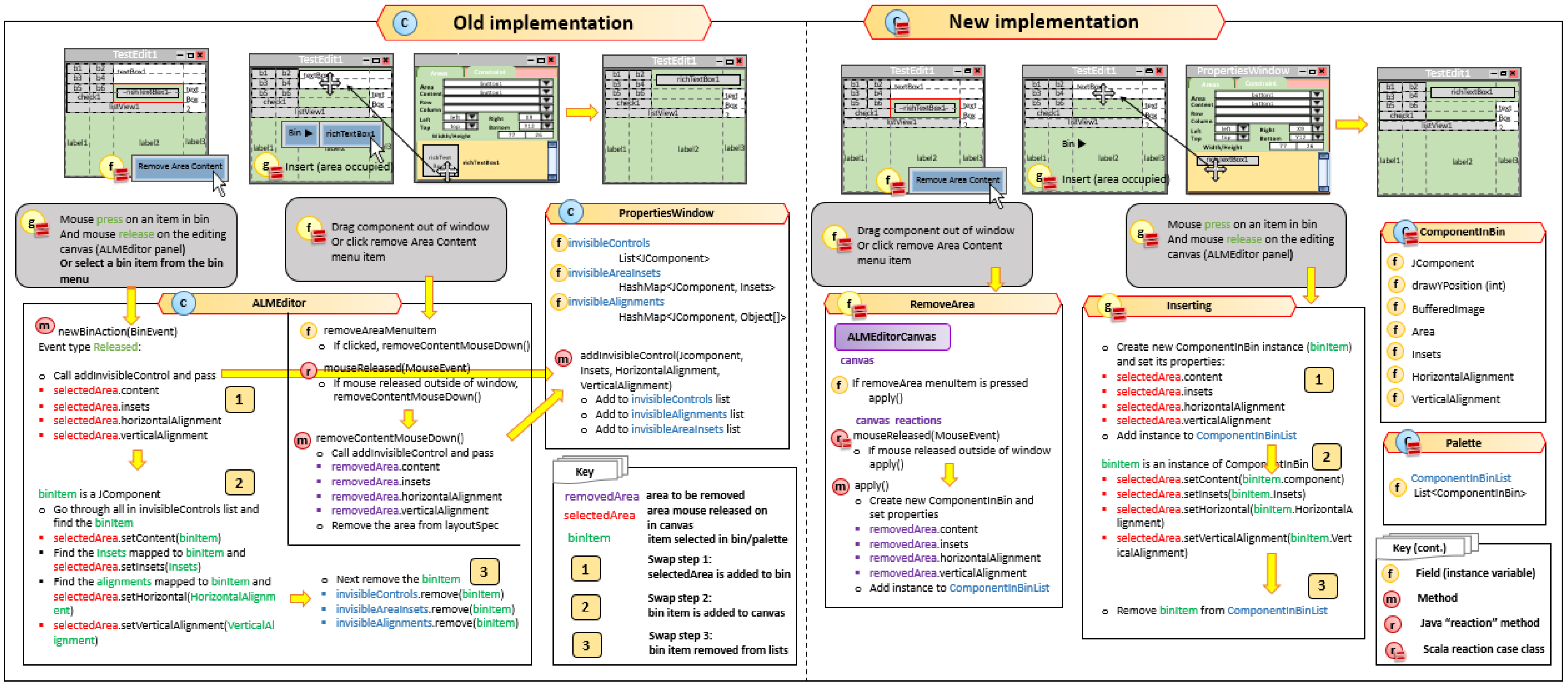

Old implementation
New implementation
Remove Area Content
Insert (area occupied)
Mouse press on an item in bin And mouse release on the editing canvas (ALMEditor panel) Or select a bin item from the bin menu
Drag component out of window Or click remove Area Content menu item
ALMEditor
newBinAction(BinEvent)
Event type Released:
Call addInvisibleControl and pass
selectedArea.content
selectedArea.insets
selectedArea.horizontalAlignment
selectedArea.verticalAlignment
binItem is a JComponent
Go through all in invisibleControls list and find the binItem
selectedArea.setContent(binItem)
Find the Insets mapped to binItem and selectedArea.setInsets(Insets)
Find the alignments mapped to binItem and selectedArea.setHorizontal(HorizontalAlignment)
selectedArea.setVerticalAlignment(VerticalAlignment)
removeAreaMenuItem
If clicked, removeContentMouseDown()
mouseReleased(MouseEvent)
If mouse released outside of window, removeContentMouseDown()
removeContentMouseDown()
Call addInvisibleControl and pass
removedArea.content
removedArea.insets
removedArea.horizontalAlignment
removedArea.verticalAlignment
Remove the area from layoutSpec
Next remove the binItem
invisibleControls.remove(binItem)
invisibleAreaInsets.remove(binItem)
invisibleAlignments.remove(binItem)
PropertiesWindow
invisibleControls List<JComponent>
invisibleAreaInsets HashMap<JComponent, Insets>
invisibleAlignments HashMap<JComponent, Object[]>
addInvisibleControl(Jcomponent, Insets, HorizontalAlignment, VerticalAlignment)
Add to invisibleControls list
Add to invisibleAlignments list
Add to invisibleAreaInsets list
Key
removedArea area to be removed
selectedArea area mouse released on in canvas
binItem item selected in bin/palette
1 Swap step 1: selectedArea is added to bin
2 Swap step 2: bin item is added to canvas
3 Swap step 3: bin item removed from lists
RemoveArea
ALMEditorCanvas
canvas
If removeArea menuItem is pressed apply()
canvas reactions
mouseReleased(MouseEvent)
If mouse released outside of window apply()
apply()
Create new ComponentInBin and set properties
removedArea.content
removedArea.insets
removedArea.horizontalAlignment
removedArea.verticalAlignment
Add instance to ComponentInBinList
Inserting
Create new ComponentInBin instance (binItem) and set its properties:
selectedArea.content
selectedArea.insets
selectedArea.horizontalAlignment
selectedArea.verticalAlignment
Add instance to ComponentInBinList
binItem is an instance of ComponentInBin
selectedArea.setContent(binItem.component)
selectedArea.setInsets(binItem.Insets)
selectedArea.setHorizontal(binItem.HorizontalAlignment)
selectedArea.setVerticalAlignment(binItem.VerticalAlignment)
Remove binItem from ComponentInBinList
ComponentInBin
JComponent
drawYPosition (int)
BufferedImage
Area
Insets
HorizontalAlignment
VerticalAlignment
Palette
ComponentInBinList List<ComponentInBin>
Key (cont.)
Field (instance variable)
Method
Java "reaction" method
Scala reaction case class

Figure 6-11 Summary of the modified bin behaviour when inserting into an already occupied area

The grey rounded boxes represent user action that is detected by the code which starts a chain of commands which either performs the removal (operation g) or insertion operation (operation f). Blue text represents the lists/maps for the bin.

<u>OPERATION F</u>

Old implementation: User action triggers the `removeContentMouseDown` method which calls `addInvisibleControl` method to add `removedArea` information to the list/maps for the bin. **New implementation:** Same as above except the `apply` method of the `Action` class is called; a new instance of the `ComponentInBin` is made (representing one bin item) and added to the list for the bin.

<u>OPERATION G</u>

Old implementation: The beginning of operation g follows steps outlined in Figure 6-7 leading up to the `newBinAction` method being called with the "Released" subEvent. Within this method, the `addInvisibleControl` method is called and `selectedArea` information is added to the list/maps for the bin (swap step 1). Next, the `selectedArea` variable is updated with the content, inset, horizontal and vertical alignment information from the `binItem` variable (swap step 2). Finally, `binItem` is removed from the list and maps (swap step 3).

New implementation: Create new `ComponentInBin` and update its properties to reflect the `selectedArea` variable (swap step 1). Next, set the conteint in the `selectedArea` variable to the one selected in bin (`binItem.component`) and update its inset,horizontal/vertical alignment information to match what the `binItem` variable had. (swap step 2), Finally, remove the `binItem` from the list (`ComponentInBinList`) (swap step 3).

6.3.4. Inserting into an Unoccupied area

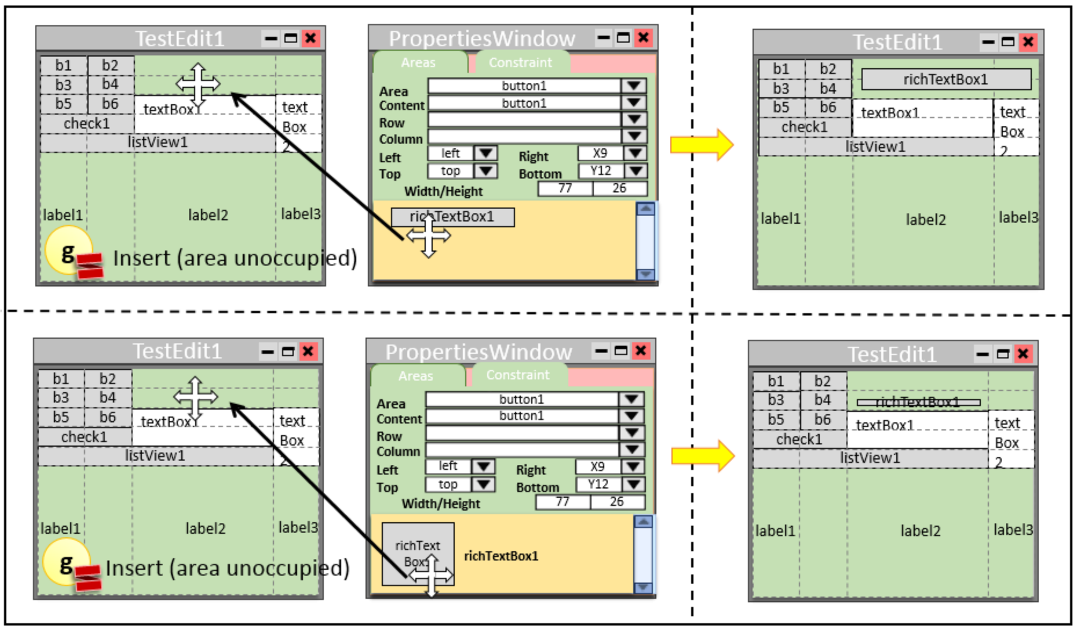


Figure 6-12 Comparing the two version of the bin feature of placing a component back into an area that has no content

The remove area editing operation (operation f) in the old implementation removes the entire area from the layout specification and also removes the top, bottom, left, right tabStops bordering the area. However this caused the overlapping of the remaining components and was dysfunctional. To simplify the operation in the new implementation, only the component is removed from the layout specification and the area remains. This is achieved with steps similar to swap steps 2 and 3 outlined in **Error! Reference source not found.**. The user clicks on a particular item in the bin, and he corresponding `ComponentInBin` instance (`binItem`) is found. The component, its horizontal/vertical alignments and inset information is extracted from the `binItem` variable and assigned to the `selectedArea` variable. Inserting into an empty area makes the component fill the entire space (Figure 6-12 top).

However the more ideal behaviour for this editing operation is “docking”: instead of having the component occupy the entire area, it is able to fill the area partially, which requires the addition of new tabStops. As an intermediary step to achieving the ideal implementation for this operation, insertion into an empty area should involve the creation of a new area. Subsequently, its borders set to be the nearest X/YTab stops. Figure 6-12 bottom illustrates what the GUI should look like after this operation.

6.3.1. Classes Diagram for the New Implementation of ALE

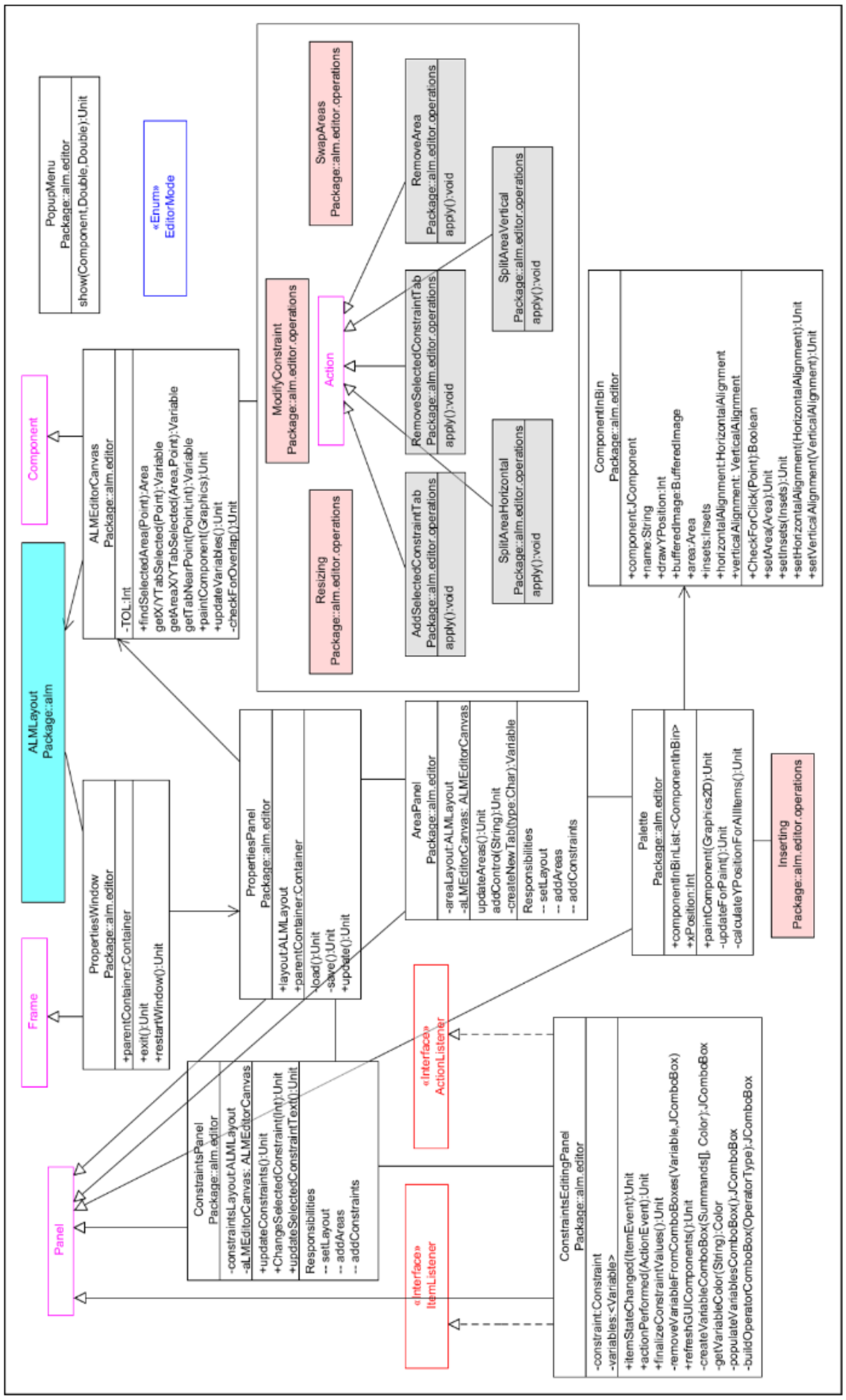
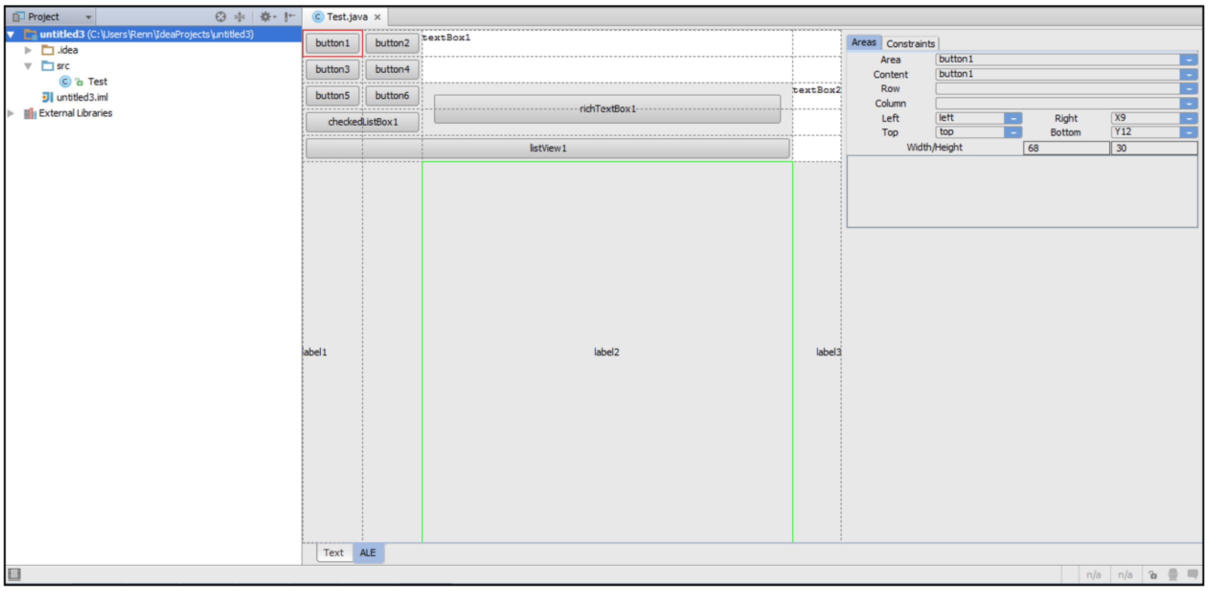

PopupMenu
Package::alm.editor
show(Component,Double,Double):Unit
«Enum»
EditorMode
Component
ALMLayout
Package::alm
Frame
Panel
ALMEditorCanvas
Package::alm.editor
-TOL:Int
+findSelectedArea(Point):Area
getX/YTabSelected(Point):Variable
getAreaXYTabSelected(Area,Point):Variable
getTabNearPoint(Point,int):Variable
+paintComponent(Graphics):Unit
+updateVariables():Unit
-checkForOverlap():Unit
PropertiesWindow
Package::alm.editor
+parentContainer:Container
+exit():Unit
+restartWindow():Unit
PropertiesPanel
Package::alm.editor
+layout:ALMLayout
+parentContainer:Container
-load():Unit
-save():Unit
+update():Unit
ConstraintsPanel
Package::alm.editor
-constraintsLayout:ALMLayout
-aLMEditorCanvas: ALMEditorCanvas
+updateConstraints():Unit
+ChangeSelectedConstraint(Int):Unit
+updateSelectedConstraintText():Unit
Responsibilities
-- setLayout
-- addAreas
-- addConstraints
SwapAreas
Package::alm.editor.operations
ModifyConstraint
Package::alm.editor.operations
Action
RemoveArea
Package::alm.editor.operations
apply():void
RemoveSelectedConstraintTab
Package::alm.editor.operations
apply():void
SplitAreaVertical
Package::alm.editor.operations
apply():void
Resizing
Package::alm.editor.operations
AddSelectedConstraintTab
Package::alm.editor.operations
apply():void
SplitAreaHorizontal
Package::alm.editor.operations
apply():void
AreaPanel
Package::alm.editor
-areaLayout:ALMLayout
-aLMEditorCanvas: ALMEditorCanvas
updateAreas():Unit
addControl(String):Unit
-createNewTab(type:Char):Variable
Responsibilities
-- setLayout
-- addAreas
-- addConstraints
ComponentInBin
Package::alm.editor
+component:JComponent
+name:String
+drawYPosition:Int
+bufferedImage:BufferedImage
+area:Area
+insets:Insets
+horizontalAlignment:HorizontalAlignment
+verticalAlignment: VerticalAlignment
+CheckForClick(Point):Boolean
+setArea(Area):Unit
+setInsets(Insets):Unit
+setHorizontalAlignment(HorizontalAlignment):Unit
+setVerticalAlignment(VerticalAlignment):Unit
Palette
Package::alm.editor
+componentInBinList:<ComponentInBin>
+xPosition:Int
+paintComponent(Graphics2D):Unit
-updateForPaint():Unit
-calculateYPositionForAllItems():Unit
Inserting
Package::alm.editor.operations
«Interface»
ActionListener
«Interface»
ItemListener
ConstraintsEditingPanel
Package::alm.editor
-constraint:Constraint
-variables:<Variable>
+itemStateChanged(ItemEvent):Unit
+actionPerformed(ActionEvent):Unit
+finalizeConstraintValues():Unit
-removeVariableFromComboBoxes(Variable,JComboBox)
+refreshGUIComponents():Unit
-createVariableComboBox(Summands[], Color):JComboBox
-getVariableColor(String):Color
-populateVariablesComboBox():JComboBox
-buildOperatorComboBox(OperatorType):JComboBox
Figure 6-13 Class diagram of ALE in the new implementation

Figure 6-13 gives an overview of the important ALE classes in the new implementation. All classes are situated in the `alm.editor` package.

More information on the new implementation of ALE is in the Appendix: See Appendix E for the containment hierarchy of the windows, Appendix F for the structure of the windows and Appendix G for the flow diagram of class instantiation after switch to editing mode.

6.4. Creating an IntelliJ Plugin



Area: button1
Content: button1
Left: left
Top: top
Right: X9
Bottom: Y12
Width/Height: 68 30

Figure 6-14 The UI of the prototype IntelliJ plugin of ALE

IntelliJ IDEA a premier IDE for Java, Groovy and Scala. A plugin is a separate module that allows the extension of the IntelliJ IDEA core functionality. There are many types of plugins, including: language support (for example the Scala plugin used in this project), Version Control System integrations (e.g. CVS), code inspections and refactoring, and utility plugins. Figure 6-14 shows the appearance of the ALE IntelliJ prototype made in this study. It belongs to the custom editor type, which supports the addition of a tabbed editor to a file in IntelliJ. The ALE plugin associates an ALE editor tab for every java file: the java class file can be edited in the "Text" tab while the testing GUI and the properties window is in the "ALE" tab (currently selected in Figure 6-14). The UI is split into two parts: the left panel which contains the testing GUI (`TestEdit1`) and the right panel which contains the `PropertiesPanel`. The IntelliJ IEA Open API library included in the Plugin Development Package provides interfaces and classes for developing plugins. Implementations of the `FileEditorProvider` and `FileEditor` interfaces were used to make this prototype (collectively hereon referred to as the IntelliJ editor system).

6.4.1. IntelliJ UI Designer Plugin from the Community Edition

Since IntelliJ IDEA has an open-source UI Designer plugin, it was studied to determine how GUI editor plugins should be written for IntelliJ. At the time of writing this report, this was the only open-source UI designer plugin. The GUI of the IntelliJ UI Designer is as follows: there is a component tree and property editor in the left-side tool window, editing panel in the centre (several JPanels layered on top of each other), and palette in the right-side tool window. Components can be dragged and dropped from the palette to the desired location on to the editing panel where they can be further manipulated. The UI Designer plugin implements its own `FileTypeFactory` interface (.form file) which allows the entire visual editor to appear as a new tab. For this project, the objective was to create a simple prototype and therefore the editor system was chosen instead of the elaborate system used by the IntelliJ Designer.

6.4.2. IntelliJ Editor System

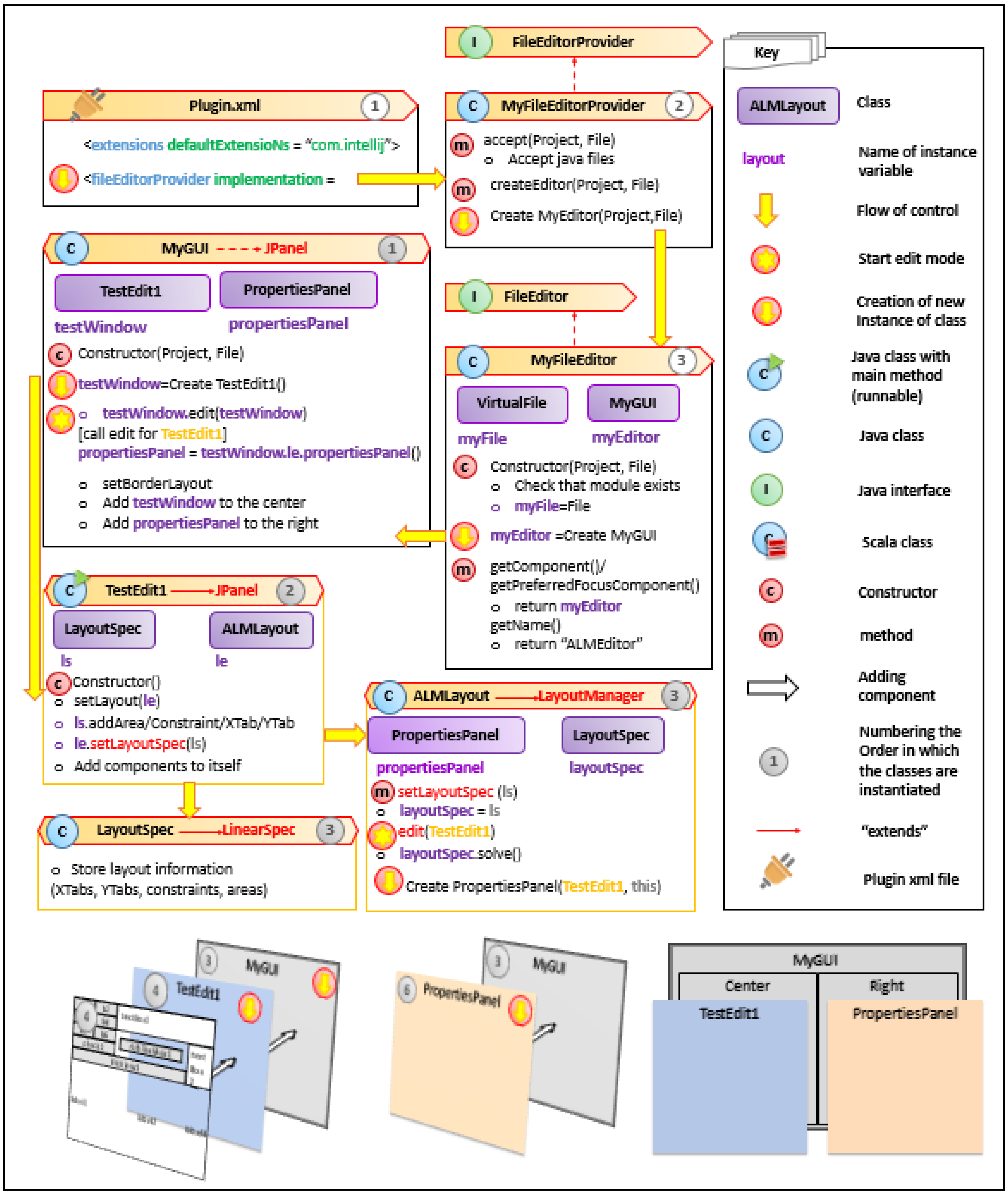


Figure 6-15 Steps for registering the editor in IntelliJ, creation of the editor panel, and start of edit mode

Below are the steps used to create the prototype plugin (summarized in Figure 6-15):

Registering fileEditorProvider as an extension in IntelliJ: `Plugin.xml` is the plugin configuration file containing information such as its version number, id, version, vendor, description, and change notes. In particular it contains xml tags to define actions and extensions. Extensions provide a means to extend the functionality of the IDEA core. The first step was to declare the `FileEditorProvider` within the `<extensions>` section in Plugin.xml by filling in the path to the `FileEditorProvider` file within the project (eg. `FileEditorProvider implementation = MyEditorProvider`).

Implementing a class that extends FileEditorProvider: This class receives two important parameters automatically from IntelliJ: a reference to the project, and a reference to the file that the editor is created for. The method accept needs to be overridden with code that checks whether the `FileEditorProvide` can create a valid `FileEditor` for the type of file (by calling the method `getFileType` on the reference to the file). Currently it is accepting Java files, but this can be changed as IntelliJ offers a number of constants in the class `StdFileTypes`. The method createEditor needs to be implemented to return an instance of a class implementing `FileEditor`. Other methods to dispose editor, read/write state, and to `getEditorTypeId/Policy` were also implemented.

Implementing a class that extends FileEditor: This class also receives the parameters project and file. Within the editor, the file is checked if it contains to a module. If not, an illegal argument exception is thrown. Whatever `JComponent` returned by `getComponent()` and `getPreferredComponent()` is the `JComponent` displayed by the editor's UI and the component in focus when the editor is opened, respectively. For both methods, an instance of the `MyGUI` class is returned therefore whatever is contained in that JPanel is displayed in the editor tab. Other methods implemented include `getName()` (returns as string for the display name of the editor), `get/setState`, `isModified/Valid`, `select/deselectNotify`, `add/removePropertyListener`, and `dispose`.

Implementing MyGUI class which extends JPanel: This class contains an instance of `TestEdit1` and an instance of `PropertiesPanel`. In the constructor, an instance of `TestEdit1` is instantiated and edit is called. The `propertiesPanel` reference within the `ALMLayout` instance associated with test window is retrieved. Next, border layout is set as the layout manager. Finally, the `TestEdit1` and `propertiesPanel` JPanels are added to the border layout center and right, thus they appear next to each other and fill up the entire space. The bottom of Figure 6-15 summarizes the containment hierarchy.

Initializing editing mode: The only difference between this version and the refactored version is that the editing mode initializes an instance of `PropertiesPanel` instead of `PropertiesWindow`, since a JPanel is required instead of a JFrame. The rest of the steps are not presented here but are illustrated in Appendix Figure 8 and Appendix Figure 9.

6.4.3. Conclusion

IntelliJ plugin development was initially difficult due to the lack of documentation compared to the extensive documentation and tutorials available for plugin development in another popular IDE, Eclipse. Therefore despite IntelliJ's advantage of having Swing support, some developers are sometimes overwhelmed by the large learning-curve involved in understanding the plugin development environment. It is hoped that experiences gained from making this prototype will help simplify the learning curve for future programmers working on the ALE plugin.

The creation of this plugin prototype is the first step in integrating ALE as a tool to be used in IntelliJ. In the future more tools and activities can be added to complement each other and make development more efficient.

Chapter 7. Evaluation

First, an evaluation of our refactoring efforts is presented. Its impact is assessed by comparing the internal quality metrics of ALE in the old vs the new implementation. In doing so we address the research question: “In how far does refactoring help to improve the quality of a complex GUI application (such as ALE)?” Also, the advantages and disadvantages of converting the code base to Scala is explored and thus we also address the research question: “Does converting a complex GUI application (such as ALE) from Java to Scala improve its code base?”

7.1. Evaluation of Program Quality

7.1.1. Internal Attributes of New Implementation of ALE

7.1.1.1. Cohesion, Coupling and Encapsulation

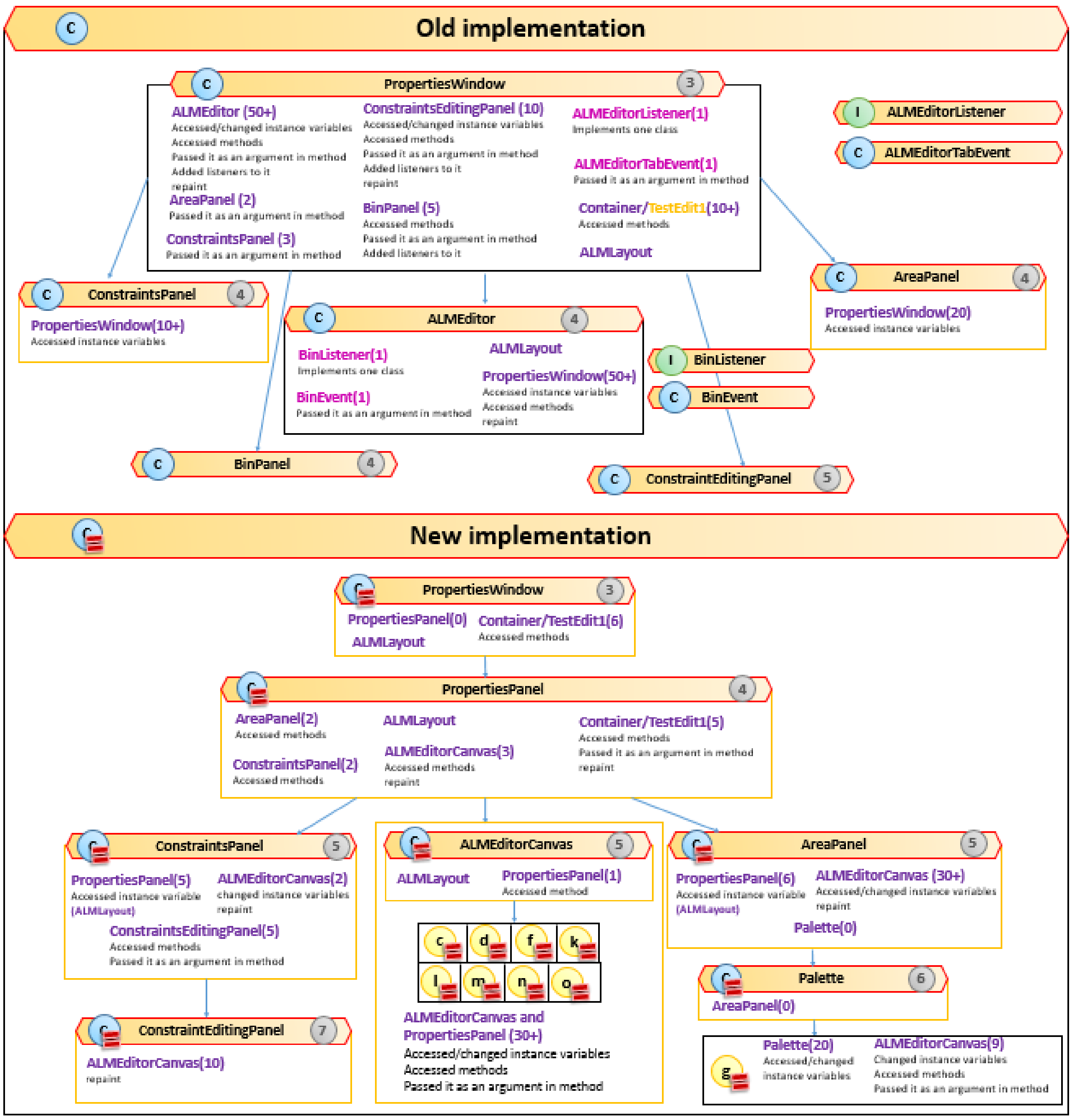


Figure 7-1 Diagram showing coupling (the number of references to other classes) within each class

Within each class box, its instance variables are highlighted in purple; the number of times it is referred to in the current class beside it in brackets; the ways in which it is referred to is listed below it (e.g. accessing methods). The references to the classes from the custom listener system are highlighted in magenta.

Due to the absence of metric calculation support for the Scala language, most of the OO metrics could not be precisely determined. It is hard to estimate whether the Coupling between Object classes (CBO) metric increased or decreased after refactoring. Here one coupling is defined as one class using methods or instance variables of another object. At the package level, import coupling was reduced by moving `PropertiesWindow` to the same subpackage as the rest of the editor classes.

Figure 7-1 is a pseudo dependency diagram showing how a particular class (illustrated as a box) interacts with other classes (class names in bold purple font) in certain ways (the ways are listed in black below the purple class names) how many times (numbers in brackets). In the old implementation, `PropertiesWindow` was coupled to eight classes and had many dependencies and can be thought of as "spaghetti code" due to its high complexity. It is also what is commonly known as a "god class" since it provided the functionality for many classes, especially for editing the canvas (i.e. features i,g,j,e in Figure 6-6). In contrast, classes like `AreaPanel`, `ConstraintsPanel` and `BinPanel` were devoid of functionality (Section 6.2.2.2). `AreaPanel` and `ConstraintsPanel` referenced JComponent instance variables in `PropertiesWindow` for setting up their layouts. `ALMEditor` was also heavily dependent on `PropertiesWindow` (indirectly retrieving it from its instance of `ALMLayout`) and also referred to the `BinListener` and `BinEvent` classes. Otherwise, `BinPanel` and `ConstraintsEditingPanel` did not reference any other classes.

The bottom half of Figure 7-1 shows that coupling is markedly reduced in the `PropertiesWindow` and `PropertiesPanel` classes, now only calling methods from `AreaPanel` and `ConstraintsPanel` or `ALMEditorCanvas` to set them up upon instantiation. `AreaPanel` and `ConstraintsPanel` only refer to `PropertiesPanel` to access its layout instance. `ConstraintPanel` refers to `ALMEditorCanvas` only to repaint and to update its `selectedConstraint` instance variable. Code in `AreaPanel` for editing area properties (feature e) still has 30 references to `ALMEditorCanvas`; most of the references are just to access the currently selected area in the canvas and modify the area's properties. Coupling between the `PropertiesPanel` and `ALMEditorCanvas` classes was reduced by removing the custom listener system described in Section 6.2.3. Low coupling is generally considered to be more beneficial because it makes individual modules easier to maintain and test [28].

The various operation classes can be thought of as plugins for the `ALMEditorCanvas`, which extend its editing capabilities. The operations heavily use methods and modify fields within `ALMEditorCanvas`, however, `ALMEditorCanvas` does not depend on the edit operation classes. These operations can be easily added/removed to increase/decrease functionality with little side effect.

Finally, different parts in the `PropertiesWindow` GUI were separated in the containment hierarchy, making its internal structure and the coupling between different parts of the GUI clearer. For example, `PropertiesWindow` includes `PropertiesPanel`, which includes `ConstraintsPanel`, which in turn includes `ConstraintEditingPanel`. This structure better reflects the visual containment of the GUI and modularizes its different parts. It achieves a greater degree of hierarchical modularization compared to before refactoring, in which the modules are layered so that the layers above are aware of the layers below it but not the other way around. Such a structure also increases encapsulation as the outer modules encase the inner modules and limit their access. This type of modularization is considered to be better quality code compared to the type of structure in before refactoring [53].

Since the metric LCOM is too difficult to calculate by hand, the cohesiveness of the program was estimated qualitatively. As Section 6.2.2.1 and 6.2.2.2 showed, the refactoring process was achieved both the separation of concerns both by placing the components within its own panel/window by separating the functionality into the logical classes (functional cohesion, where parts of a module are grouped together because they all contribute to a well-defined task). As a result, in the new implementation the fields and methods in each class are highly correlated to produce behaviour that is more cohesive. High cohesion indicates good class subdivision and reduces complexity, thereby decreasing the chance of errors arising during development [31]. Furthermore, such programs are easier to maintain because each change is localized in a single cohesive module, and each module can be easily reused. Although the degree of cohesion could not be estimated, since functional cohesion has been shown to be the most powerful form of the cohesion types [54], this provides good evidence for the effectiveness of the modularization achieved in the new implementation.

7.1.1.2. Complexity

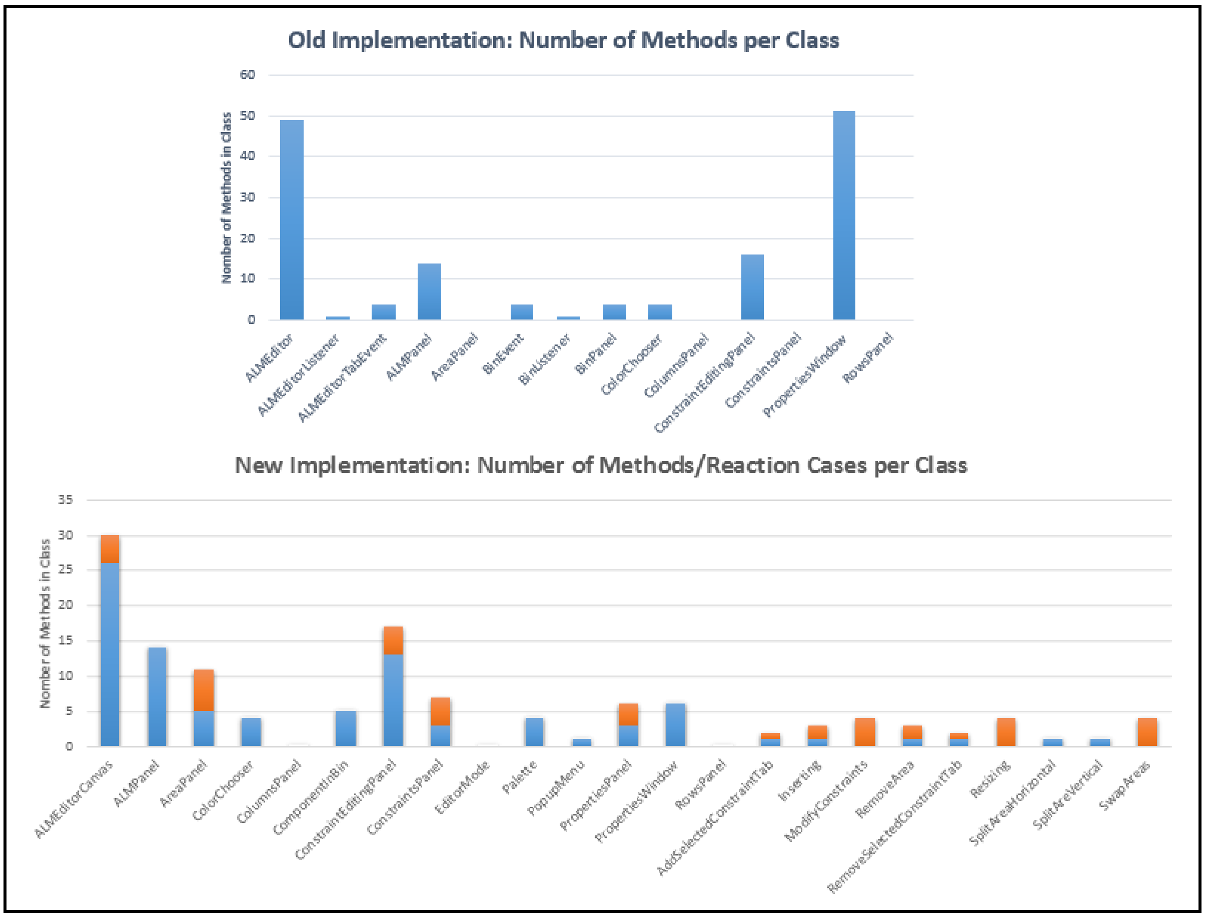


Figure 7-2 Graphs showing the number of methods per class before and after refactoring

The main quantitative measurement to estimate complexity was the number of methods since there are no tools to measure WMC for Scala classes. The total number of methods in each class was a metric that was somewhat hard to measure in the new implementation since much of the functionality was moved to Scala reactions but they are not counted as methods. To compensate, each case class in Scala was counted manually and added to the total count; the normal methods are coloured blue and the Scala cases are coloured orange in Figure 7-2. In the old implementation, `ALMEditor` and `PropertiesWindow` contained over 90% of the methods (Figure 7-2). In the new implementation, the number of methods has reduced to about half for `ALMEditor` and to about a tenth for `PropertiesWindow`. Although the exact cyclomatic complexity for a particular method could not be measured, the Scala-style checker indicated that there were several methods in the `ALMEditorCanvas` class both before and after refactoring that exceeds the recommended CC of 10. Therefore some improvement can still be made there.

Furthermore, in the new implementation there is a more uniformity in the distribution of methods across classes. This, together with the evidence for increased cohesion and decreased coupling presented in Section 7.1.1.1 indicate increased modularity- and thus a reduction in complexity. In the old implementation, the total number of methods was 148 while in new implementation, total number of methods (including the Scala cases) is 129. Reduction in number is due mostly due to removal of unused methods and integrating methods into code, which also help simplify the code and increase comprehensibility.

7.1.1.3. Inheritance

In total, the program before refactoring extended eight interfaces and inherited from five. After refactoring, it extends five interfaces and inherits from five, plus five operation classes that implement the `apply` method from the abstract Action class. Extending a smaller number of interfaces does not reduce the functionality since all the code within the `eventXhandlerMethods` have been moved into reaction cases after conversion to Scala. The inheritance depth of the existing classes generally did not change, as now they are inheriting from Scala swing instead of Java swing. From examining Figure 3-1 we can see that most components has three ancestors in Java they are `JComponent`, `Container`, and `Object`; and most components has four ancestors Scala they are `Component`, `UIElement`, and `Proxy`, and `Any`. A class with deeper inheritance hierarchy is regarded as more complex since it inherits a larger number of methods and its behaviour becomes hard to predict [55]. It also requires greater planning and design time since more methods and classes are involved [27]. The edit operation classes increases the total number of inheriting classes in the new implementation, inheriting the `apply` method from the `Action` class. However we believe this actually simplifies code and avoids the need to create an anonymous `Action` class in the constructor of the new `MenuItems`. Also, since the `Action` class directly inherits from the root class `AnyRef`, it should not complicate the code to a great extent.

7.1.1.4. Number of Classes, Lines of Code (LOC), Comment Percentage

Table 4 summarizes some general statistics for the classes in the editor package before and after refactoring. Firstly, the number of classes increased from 14 to 23. This is due to the addition of the edit operation classes, `ComponentInBin` class, `EditorMode` enum class, and `PopupMenu` class. As mentioned previously, the addition of operation classes increases the cohesiveness and reusability while simplifying code due to the clear separation of components and functionality. The `ComponentInBin` class helps with the simplification of the bin functionality (Section 6.3.3), the Enum class also achieves better separation of concern while the `PopupMenu` class is required as a custom wrapper class. Therefore we believe that this increase in the number of classes achieves better design while maintaining same (or better) functionality.

Secondly, LOC was measured in several ways, including counting physical lines of code with/without blank or commented lines. Figure 7-3 shows that the LOC per class follows a similar trend to and methods per class (Figure 7-2), in that the numbers are more uniform in the new implementation. The class with the greatest initial LOC (`ALMEditor`) also had the greatest reduction, to about half of its original amount in the new implementation. These reductions can be attributed to removal of redundant code, moving the variable declaration and instantiation to be on the same line, and the more concise Scala syntax. It is generally accepted that between projects designed to achieve the same functionality, projects with less LOC is easier to maintain and understand [34, 56]. Both the total LOC and code LOC has reduced significantly in the new implementation even in the face of slightly increased functionality, suggesting that the refactored code is of higher quality.

Lastly, the comment percentage was calculated by dividing total number of comments by the total lines of code (without the number of blank lines), as is the usual convention (Table 4). However due to the limitation of the metric calculation tool used, the on-line comments are not counted (so only the stand-alone comments are included in the statistics. It has been found by SATC that a comment percentage of approximately 30% is most effective, and assists in attributes of comprehensibility, reusability and maintainability. The comment percentage in the new implementation is higher at 31% and is therefore likely better documented.

Statistic/Metric	Old Implementation	New Implementation
Number of Java/Scala classes	14/0	5/18
Blank LOC	538	225
% blank	10%	6%
Comment LOC	1215	1093
% comment	26%	31%
Code LOC	3501	2419
% code	67%	65%
Total LOC	5254	3737

Table 4 Summary statistics for the ALE in the old vs the new implementation

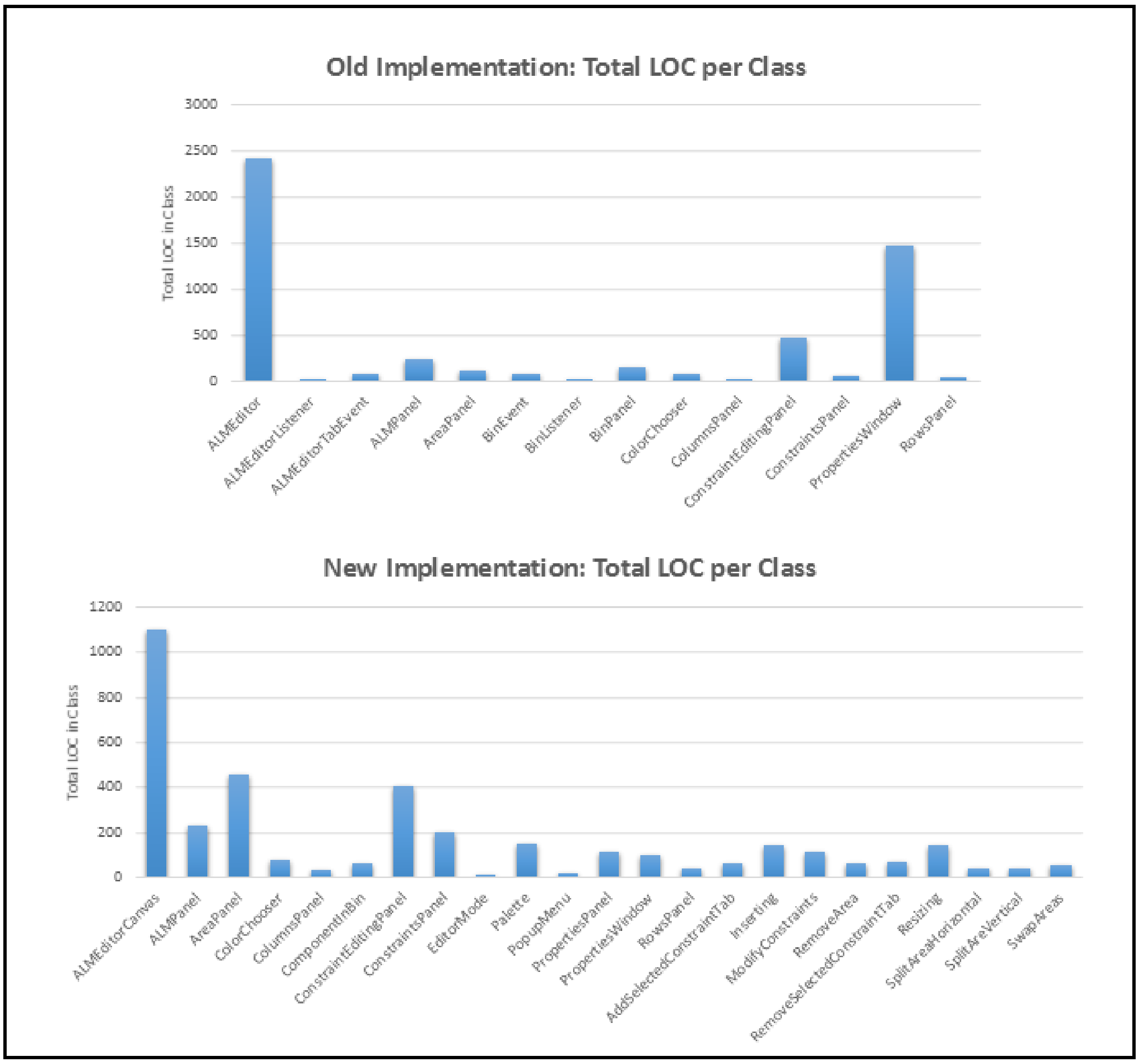


Figure 7-3 Graphs showing total LOC per class before and after refactoring

7.1.2. Influence on Overall Quality

Conclusions from this section about software quality are supported by research linking certain internal attributes/metrics to external quality attributes (Section 2.1).

The cohesiveness of the code has increased and coupling has decreased, which indicates improved maintainability since each class can be maintained separately from another. It also improves readability and comprehensibility as each section is logically organized. It also increases the reusability of the code by separating it into operation classes (both functionally and the components) with minimal coupling, everything belonging to a particular operation is together, so if in the future the code needs to be used somewhere, it be done easily. It is also more extensible, due to the editing operation classes which act as a type of “plugin”, more functionality can easily be added in the future.

Complexity has reduced due to a number of reasons: improved cohesiveness, reduced LOC and number of methods, the use of the simpler Scala syntax and constructs (e.g. collection transformations can be done in as few as one line), and better documentation of the source code through commenting. This is beneficial to many attributes, including adaptability, maintainability, comprehensibility, and reusability.

Usability has improved after modifying the old implementing features. The appearance of the bin items in the old implementation did not allow the user to see how large the component was before adding to the bin. The use of buffered images in the new implementation allows the user to see the exact appearance and size of the GUI component in the bin and makes making reinserting the bin item more intuitive. In addition, removing bin menu makes the editing less confusing. Removing by dragging and dropping outside the window is more intuitive and should be the only action needed.

In conclusion, refactoring ALE has improved a number of quantitative internal measures and we believe this improvement in quality can also be percieved on the external level. In particular, the following attributes seem to have improved: maintainability comprehensibility, readability, reusability, adaptability, extensibility and usability. These findings are in line with most previous work that claims that refactoring is beneficial for software development and quality (Section 2.1).

7.2. Evaluation of the Conversion to Scala

Firstly, the evidence provided in Section 7.1 above indicates that internal quality attributes have improved. In particular the total LOC and SLOC has reduced significantly. A large part of this improvement is due to Scala Swing’s more concise syntax. There is no need for anonymous listener classes as the appropriate code for each event can be defined in the less verbose pattern matching statements (Section 6.1.3). Also, the collections transformations negates the need for explicit for-loops (Section 6.1.4).

Scala’ GUI hierarchy categorises the Components more finely compare to Java’s system and is ideal for creating custom components. They allow for the mixing and matching of general constructs to achieve a more complex one. The custom class can inherit from a number of traits to get the concrete members it require for its particular purpose. It also has a more powerful collections system with transformation methods that can manipulate collections without the need to explicitly define loops. The use of case classes and partial functions is a powerful system for event handling and removes the need for the complex task to define anonymous classes and override different `eventXHandler` methods as in Java. Moreover, the reactions handler block mean it is possible to define all handler reactions in one place, simplifying the code and improving readability. Alternatively, handler code can be separated if needed due to the simplicity of installing reactions in Scala, which benefits the separation of concerns (see Section 6.2.2). It is primarily through the creation of the edit operations that Cohesion, Coupling was improved and complexity reduced.

It is easy to transition to Scala from a Java background due to its complete interoperability with Java. A beginner can always start with Java code and migrate slowly to using Scala constructs. However one disadvantage of using Scala is the rather steep learning curve, further hindered by the difficult API and lack of code samples compared to Java. It was especially difficult adapting the existing simple code examples to a complex GUI application. We found that the command line and interactive interpreters, while useful for learning concepts, did not help in providing clues on how to solve problems in a complex GUI application. Our experiences are similar to those described by the subjects in [40] which found that Scala is more difficult to understand and the documentation lacking.

Furthermore, there is little support for the IntelliJ Scala plugin and the initial step in converting Java to Scala yielded many errors (see appendix C) which were difficult to solve. Scala is also disadvantageous for the fact that some missing features in Scala means that javax.swing peer must be called and even then sometimes the peer member does not exist and a wrapper class has to be written manually.

Overall we believe that converting the code base of ALE to Scala produced many benefits for the code base, improving its maintainability and comprehensibility especially in conjunction with the correct refactoring techniques. Therefore it may be summarized that in general that complex GUI applications written in Java will achieve benefits by conversion to Scala. No studies to date have evaluated programmer experience with the Scala language in the context of GUI programs. It is hoped that these conclusions will help programmers in the future who work on complex GUI applications like ALE.

7.3. Threats to Validity

The quality evaluation demonstrated that cohesion has increased, while coupling, complexity, and LOC have reduced and inheritance has remained the same. These changes have led to improved characteristics and an overall better quality program as described above. It is obvious that the ability to generalize from one particular case study is extremely limited. Since validation was based on the work of one programmer on one short-term project, the findings may not be applicable in a different context. Similarly, the evaluated advantages of converting the code base to Scala is based on one programmer's experiences and is a small, subjective sample and may not be representative of the general opinion. Moreover, the long-term effects of ALE's refactoring, such as whether it increases development speed or decreases bugs in the program, still remain to be seen.

The experimental design of the evaluation contains a few flaws. The study was not completely controlled: the refactored code also contains code improving or adding functionality, so not all changes can be attributed to refactoring or conversion to Scala. Also, since both the conversion and refactoring were applied to ALE simultaneously, the positive effect of one cannot be separated from the other. The ideal method to answer the question "in how far refactoring improves quality" would be by comparing a program before and after refactoring where changes are due to refactoring techniques alone, as was the method of previous studies (Section 2.2). To answer the question, "does converting to Scala improve program quality?", better experimental design would involve a more extensive empirical study like the one in [40] with two groups working on the same program in Java and Scala for a more valid comparison. Consequently, more specific questions such as whether it is easier to perform refactoring in Scala compared to Java, whether it is easier to implement new features in Scala compared to Java, can be assessed.

Furthermore, the identification of problematic locations in code that would benefit from refactoring was based on human intuition much like the notion of "bad smells" as described in [14]. The ability of one programmer to detect such locations may be limited and tool support is often necessary. In the future, tools could be used to aid decisions on refactoring such as the one described in [23] which detects potential design flaws by analysing metrics.

Lastly, the object-oriented metrics used in this study to represent internal quality attributes may not be sufficient, since there are strong functional programming capabilities that have to be taken into account when measuring the quality of Scala programs. Functional programming metrics for complexity have been explored in the literature [57, 58]. More recently, metric sets have been suggested for measuring modularity in a functional programming system and validated for Scala Systems [59, 60]. These should be incorporated into future evaluations to assess the extent of good functional programming practices.

Chapter 8. Future Directions

Currently, ALE Java Version does has not implemented all the edit operations and features which exist in the original C++ version [2]. Notably, the rows and columns abstraction is missing and some of the edit operations are not fully functional. As described in Section 6.3.4, inserting into an unoccupied area fills the entire area instead of docking the component that is smaller than the area by snapping it to a XTab or YTab. Furthermore, it does not support the filling of empty "gaps" after the removal of a component. It is also missing the ability to specify the preferred, minimum, maximum sizes of areas and soft constraints.

Unlike the original ALE which, always ensures the areas are non-overlapping, this criteria is not always satisfied in the current implementation. Other missing functionality and bugs are outlined in Appendix K.

Instead of having to define all components of the GUI in the source code of an application, ALE should support the addition of new GUI components from a palette onto the canvas. The ability to support custom components also needs to be included, as well as the ability to easily import, export and share GUIs. The work on the initial IntelliJ prototype should be expanded upon. For example, the plugin needs to support better integration with the IntelliJ IDEA, such as allowing for the editing and use of a created GUI in an IntelliJ project.

Chapter 9. Conclusion

This paper has presented ALE as a case study for investigating questions relating to refactoring and the use of Scala. GUI programs such as ALE can become complex and hard to maintain over time. Refactoring is a technique that has been shown in literature to improve software quality. However there is a general lack of empirical evaluation done about the effect of refactoring on a complex GUI application such as ALE. Scala Swing is claimed to help alleviate most of the disadvantages of Java Swing for writing GUI applications. Our data corroborate the previous refactoring studies and show that refactoring improves the internal metrics: cohesion, coupling, and complexity. It also seems to overall software understandability, reusability, extensibility and maintainability.

We have also demonstrated a method for increasing the separation of concerns in a complex GUI application such as ALE by using the reactions member in Scala's event handling system. We found that Scala Swing is indeed more powerful than Java Swing, providing the programmers more flexibility by offering Scala's enhanced API while still providing the option to use the underlying Java peer elements. Our experiences with Scala mostly support the claims about Scala's benefits. We found it to be more concise and clearer than Java, especially in the use of the functional style, e.g. the pattern matching and collection transformation methods. In addition event handling in Scala was found to be more intuitive and less verbose than Java. Therefore the conversion to Scala seem to aid software understandability, readability and maintainability.

We have also added functionality to ALE and modified some of its exiting functionality, however there are still missing edit operations that needs to be added in the future. In addition, it can be improved by allowing for the creation of GUIs from scratch and supporting import, export and sharing options. Finally, we developed a plugin for IntelliJ IDEA by implementing its editor interfaces. This is an import first step to achieve full integration of ALE into IntelliJ.

Bibliography

[1] C. Lutteroth, R. Strandh, and G. Weber, "Domain specific high-level constraints for user interface layout," *Constraints,* vol. 13, pp. 307-342, 2008.

[2] C. Zeidler, C. Lutteroth, G. Weber, and W. Stürzlinger, "The Auckland layout editor: an improved GUI layout specification process," in *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, 2012, pp. 103-103.

[3] D. Draheim, C. Lutteroth, and G. Weber, "Graphical user interfaces as documents," in *Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: design centered HCI*, 2006, pp. 67-74.

[4] J. Kim and C. Lutteroth, "Multi-platform document-oriented guis," in *Proceedings of the Tenth Australasian Conference on User Interfaces-Volume 93*, 2009, pp. 27-34.

[5] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: a comprehensive step-by-step guide*: Artima Inc, 2008.

[6] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 126-139, 2004.

[7] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004, pp. 144-151.

[8] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 576-585.

[9] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," in *Reuse of Off-the-Shelf Components*, ed: Springer, 2006, pp. 287-297.

[10] M. Bruntink and A. van Deursen, "An empirical study into class testability," *Journal of Systems and Software,* vol. 79, pp. 1219-1232, 2006.

[11] F. Dandashi, "A method for assessing the reusability of object-oriented code using a validated set of automated measurements," in *Proceedings of the 2002 ACM symposium on Applied computing*, 2002, pp. 997-1003.

[12] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 897-910, 2005.

[13] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on,* vol. 28, pp. 4-17, 2002.

[14] M. Fowler, *Refactoring: improving the design of existing code*: Addison-Wesley Professional, 1999.

[15] K. Stroggylos and D. Spinellis, "Refactoring--Does It Improve Software Quality?," in *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, 2007, pp. 10-10.

[16] B. Du Bois and T. Mens, "Describing the impact of refactoring on internal program quality," in *International Workshop on Evolution of Large-scale Industrial Software Applications*, 2003, pp. 37-48.

[17] B. Du Bois, S. Demeyer, and J. Verelst, "Does the Refactor to Understand Reverse Engineering Pattern Improve Program Comprehension?," in *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 334-343.

[18] B. Geppert, A. Mockus, and F. Robler, "Refactoring for Changeability: A way to go?," in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, pp. 10 pp.-13.

[19] D. Wilking, U. F. Kahn, and S. Kowalewski, "An Empirical Evaluation of Refactoring," *e-Informatica,* vol. 1, pp. 27-42, 2007.

[20] R. Leitch and E. Stroulia, "Assessing the maintainability benefits of design restructuring using dependency analysis," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, 2003, pp. 309-322.

[21] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *OOIS 2001*, ed: Springer, 2001, pp. 113-122.

[22] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*, ed: Springer, 2008, pp. 252-266.

[23] L. Tahvildari and K. Kontogiannis, "Improving design quality using meta-pattern transformations: a metric-based approach," *J. Softw. Maint. Evol.: Res. Pract,* vol. 16, pp. 331-361, 2004.

[24] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and software technology,* vol. 51, pp. 1319-1326, 2009.

[25] J. C. Coppick and T. J. Cheatham, "Software metrics for object-oriented systems," in *Proceedings of the 1992 ACM annual conference on Communications*, 1992, pp. 317-322.

[26] R. Carapuça, "Candidate metrics for object-oriented software within a taxonomy framework," *Journal of Systems and Software,* vol. 26, pp. 87-96, 1994.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on,* vol. 20, pp. 476-493, 1994.

[28] L. C. Briand, J. W. Daly, and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems," *Software Engineering, IEEE Transactions on,* vol. 25, pp. 91-121, 1999.
[29] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on,* pp. 308-320, 1976.
[30] F. B. e Abreu, "The MOOD metrics set," in *proc. ECOOP*, 1995, p. 267.
[31] N. E. Fenton and S. L. Pfleeger, *Software metrics: a rigorous and practical approach*: PWS Publishing Co., 1998.
[32] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. J. Lucena, "On the modularity of software architectures: A concern-driven measurement framework," in *Software Architecture*, ed: Springer, 2007, pp. 207-224.
[33] I. Jacobson, *Object-oriented software engineering: a use case driven approach*: Pearson Education India, 1992.
[34] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*: Prentice-Hall, Inc., 1994.
[35] ISO/IEC, " ISO/IEC 9126 - Information Technology - Software Product Evaluation/ Quality Characteristics and Guidelines for Their Use," *International Orgaization for Standardization/ International Electrotechnical Commission,* 1996.
[36] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on,* vol. 22, pp. 751-761, 1996.
[37] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, *et al.*, "new Scala () instance of Java: a comparison of the memory behaviour of Java and Scala programs," *ACM SIGPLAN Notices,* vol. 47, pp. 97-108, 2013.
[38] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *ACM SIGPLAN Notices*, 2006, pp. 169-190.
[39] M. Denti and J. K. Nurminen, "Performance and energy-efficiency of Scala on mobile devices," in *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2013 Seventh International Conference on*, 2013, pp. 50-55.
[40] V. Pankratius, F. Schmidt, and G. Garretón, "Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 123-133.
[41] R. Hundt, "Loop recognition in c++/java/go/scala," *Proceedings of Scala Days,* vol. 2011, 2011.
[42] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, *et al.*, "An overview of the Scala programming language," Citeseer2004.
[43] Scala-lang.org. (2009, 01/24). *Scala in the Enterprise*. Available: http://www.scala-lang.org/old/node/1658
[44] B. Venners, "Twitter on Scala," *Artima Developer. http://www. artima. com/scalazine/articles/twitter_on_scala. html. Retrieved,* pp. 06-17, 2009.
[45] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys (CSUR),* vol. 30, pp. 123-169, 1998.
[46] K. Davis, Y. Smaragdakis, and J. Striegnitz, "Multiparadigm programming with object-oriented languages," in *Object-Oriented Technology ECOOP 2002 Workshop Reader*, 2002, pp. 154-159.
[47] M. Schinz and P. Haller, "A Scala tutorial for Java programmers," ed: December, 2007.
[48] C. J. Tauro and K. Dhanashree, "Expressing Object-Oriented Thoughts Functionally," *International Journal of Computer Applications,* vol. 65, 2013.
[49] D. Ghosh and S. Vinoski, "Scala and lift functional recipes for the web," *Internet Computing, IEEE,* vol. 13, pp. 88-92, 2009.
[50] M. S. Bhat, D. G. Nair, D. Bansal, and J. Vaishnavi, "Data structure based performance evaluation of emerging technologies—A comparison of Scala, Ruby, Groovy, and Python," in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, 2012, pp. 1-5.
[51] M. C. Lewis, *Introduction to the art of programming using Scala*: CRC Press, 2012.
[52] H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re) shape evolving software," *Communications of the ACM,* vol. 44, pp. 43-50, 2001.
[53] M. P. D. Chowhan and S. Dhande, "Comparative Study of the Modularization Techniques used for an OOS System to Achieve Quality," *International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE),* vol. 2, pp. pp: 181-185, 2013.
[54] S. McConnell, *Code complete*: O'Reilly Media, Inc., 2004.
[55] D. A. Gustafson and B. Prasad, *Properties of software measures*: Springer, 1992.
[56] R. V. Hudli, C. L. Hoskins, and A. V. Hudli, "Software metrics for object-oriented designs," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*, 1994, pp. 492-495.
[57] K. Van den Berg and P. van den Broek, "Static analysis of functional programs," *Information and Software Technology,* vol. 37, pp. 213-224, 1995.
[58] C. Ryder, "Software Measurement for Functional Programming," Citeseer, 2004.
[59] M. N. Gubitosi, M. Basavaraju, and A. M. Asadullah, "Metrics for Measuring the Quality of Modularization of Scala Systems," in *APSEC Workshops*, 2012, pp. 9-16.

[60] B. Muddu, A. Asadullah, V. Bhat, and S. Padmanabhuni, "Metrics for Modularization Assessment of Scala and C# Systems," 2013.

Appendices

Appendix A - Tools Used in this Study

1) IntelliJ IDEA was used for developing the project in Java (and with the Scala plugin for developing in Scala)
2) All images were drawn from scratch using Microsoft Powerpoint by the author(except for Figure 1-1 and Figure 6-14, which are screenshots). The class diagrams were drawn with Umlet 12.2.
3) Basic statistics (lines of comments, lines of code, total lines of code) were calculated using cloc statistic tool (compatible with both Java and Scala)
4) The rest of the statistics were either calculated from existing statistics (% of code, % of comments) or manually counted (number of methods)
5) Scala-style was briefly used to detect potential problems in the code and to gauge the CC metric
6) NotePad++ in association with word was used to display properly formatted code in the text.
   First, the Scala syntax highlighting has to be activated by downloading the scala distribution
   Second, copy and paste userDefineLang.xml from scala-dist/tool-support/src/notepad-plus/userDefineLang.xml to %AppData% \Roaming\Notepad++ and restart
   Third, in word and in the main document go to insert> object > object > openDocument text (this will create another document in a new window: this contains the code snippet)
   Fourth, in NotePad++, open the java/scala file containing the code snippet you want and go to plugins>NppSupport> copy all formats to clipboard.
   Finally, press paste back in the new document created in word, edit the snippet and press save
   The snippet should show up in the main document

Appendix B - Setting up Scala

1. Download the Scala plugin by going to File>Settings>Plugins> Browse repositories>Search and select the Scala plugin > press “download” and restart
2. Also download Scala Imports Organizer plugin the same way > click close > OK> will be prompted to restart IntelliJ to activate the plugins > Restart
3. Project Settings > Project > Project SDK> Click “New” and select the directory to the Java JDK for example: C:\Program Files\Java\jdk1.7.0_45
4. Download Scala from http://www.Scala-lang.org/download/ (version used in this project is 2.10.2.
5. File > Project structure > platform settings> Global library: Click the “add” button > Select Java> go to the folder containing the Scala download >select its “lib” folder and click “ok”. Now the Global library list should have “C:\Scala-2.10.2\lib” within it
6. Similarly in File > Project structure > Project settings > libraries, do the same as above (click add button and add the lib folder from the Scala download)
7. File > Project structure > platform settings> Modules > press the “+” arrow and add the Scala facet. Make sure compiler library is set to be the download version of Scala (in this project version 2.10.2) and the compile order is mixed. Also make sure that the module SDK is set to be IDEA IC-133.331
8. File > Project structure > platform settings> Modules > make sure compiler library is set to be the download version of Scala (in this project version 2.10.2) and the compile order is mixed.

Clicking the run button should build and compile the project too and IntelliJ will say “using an external compiler”

Appendix C - Errors Encountered when Converting Java to Scala

Error 1:

```
Scala: no-symbol does not have an owner
Scala: uncaught exception during compilation: Scala.reflect.internal.FatalError
Scala: Error:
    while compiling: C:\Users\Renn\Desktop\ALMfolder\src\alm\other\ALMEditorCanvas.Scala
       during phase: global=explicitouter, atPhase=erasure
    library version: version 2.10.3
    compiler version: version 2.10.3
 reconstructed args: -classpath C:\Users\Renn\Desktop\ALMfolder\output;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\deploy.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\Javaws.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\jce.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\jfxrt.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\management-agent.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\plugin.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\resources.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\rt.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\ext\access-bridge-64.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\ext\jaccess.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\ext\localedata.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\ext\sunec.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\ext\sunjce_provider.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\ext\sunmscapi.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\ext\zipfs.jar;C:\Users\Renn\Desktop\ALMfolder\libs\lpsolve55j.jar;C:\Users\Renn\Desktop
\ALMfolder\libs\commons-math3-3.0.jar;C:\Users\Renn\Desktop\ALMfolder\libs\matlabcontrol-
4.0.0.jar;C:\Users\Renn\Desktop\ALMfolder\libs\opt4j-
2.7.jar;C:\Users\Renn\Desktop\ALMfolder\libs\pdstore.jar;C:\Program Files (x86)\JetBrains\IntelliJ IDEA Community
Edition 13.0\lib\junit-4.10.jar;C:\Program Files (x86)\Scala\lib\akka-actors.jar;C:\Program Files
(x86)\Scala\lib\diffutils.jar;C:\Program Files (x86)\Scala\lib\jline.jar;C:\Program Files (x86)\Scala\lib\Scala-actors-
migration.jar;C:\Program Files (x86)\Scala\lib\Scala-actors.jar;C:\Program Files (x86)\Scala\lib\Scala-
compiler.jar;C:\Program Files (x86)\Scala\lib\Scala-partest.jar;C:\Program Files (x86)\Scala\lib\Scala-
reflect.jar;C:\Program Files (x86)\Scala\lib\Scala-swing.jar;C:\Program Files (x86)\Scala\lib\Scalap.jar;C:\Program Files
(x86)\Scala\lib\typesafe-config.jar -bootclasspath C:\Program Files\Java\jdk1.7.0_45\jre\lib\resources.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\rt.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\sunrsasign.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\jce.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.7.0_45\jre\lib\jfr.jar;C:\Program
Files\Java\jdk1.7.0_45\jre\classes;C:\Program Files (x86)\Scala\lib\Scala-library.jar
 last tree to typer: Ident(ex)
            symbol: value ex (flags: <triedcooking>)
  symbol definition: val ex: Scala.runtime.NonLocalReturnControl[Boolean @unchecked]
               tpe: ex.type
      symbol owners: value ex -> method checkForOverlap -> class ALMEditorCanvas -> package other
     context owners: value e -> method mousePressed -> anonymous class $anon -> constructor ALMEditorCanvas ->
class ALMEditorCanvas -> package other
== Enclosing template or block ==
DefDef( // override def mousePressed(e: Java.awt.event.MouseEvent): Unit
 <method> override
 "mousePressed"
 []
 // 1 parameter list
 ValDef( // e: Java.awt.event.MouseEvent
    <param>
    "e"
    <tpt> // tree.tpe=Java.awt.event.MouseEvent
    <empty>
 )
 <tpt> // tree.tpe=Unit
 Try( // tree.tpe=Unit
    Apply( // final def removeContentMenuItem_MouseDown(): Unit in class ALMEditorCanvas, tree.tpe=Unit
     ALMEditorCanvas.this."removeContentMenuItem_MouseDown" // final def
removeContentMenuItem_MouseDown(): Unit in class ALMEditorCanvas, tree.tpe=()Unit
     Nil
    )
    CaseDef( // tree.tpe=Unit
     Bind( // val e1: Exception, tree.tpe=Exception
       "e1"
       Typed( // tree.tpe=Exception
         "_" // tree.tpe=Exception
         <tpt> // tree.tpe=Exception
       )
     )
     Apply( // def printStackTrace(): Unit in class Throwable, tree.tpe=Unit
       "e1"."printStackTrace" // def printStackTrace(): Unit in class Throwable, tree.tpe=()Unit
       Nil
     )
    )
 )
)
```
Fix 1:
Caused by the presence of inner classes, fixed by removing the inner classes

Error 2:
warning: [options] bootstrap class path not set in conjunction with -source 1.6

Fix 2:
File>Project structure> project> change project language level to 7.0
Useful links:
http://docs.oracle.com/Javase/7/docs/technotes/tools/solaris/Javac.html#xlintwarnings
https://blogs.oracle.com/darcy/entry/bootclasspath_older_source
http://stackoverflow.com/questions/15882586/bootstrap-class-path-not-set

Error 3:
```
Exception in thread "main" Java.lang.
UnsupportedClassVersionError: aim/TestEdit1 : Unsupported major.minor version 51.0
    at Java.lang.ClassLoader.defineClass1(Native Method)
    at Java.lang.ClassLoader.defineClassCond(ClassLoader.Java:631)
    at Java.lang.ClassLoader.defineClass(ClassLoader.Java:615)
    at Java.security.SecureClassLoader.defineClass(SecureClassLoader.Java:141)
    at Java.net.URLClassLoader.defineClass(URLClassLoader.Java:283)
    at Java.net.URLClassLoader.access$000(URLClassLoader.Java:58)
    at Java.net.URLClassLoader$1.run(URLClassLoader.Java:197)
    at Java.security.AccessController.doPrivileged(Native Method)
    at Java.net.URLClassLoader.findClass(URLClassLoader.Java:190)
    at Java.lang.ClassLoader.loadClass(ClassLoader.Java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.Java:301)
    at Java.lang.ClassLoader.loadClass(ClassLoader.Java:247)
    at Java.lang.Class.forName0(Native Method)
    at Java.lang.Class.forName(Class.Java:171)
    at com.IntelliJ.rt.execution.application.AppMain.main(AppMain.Java:113)
```
Fix 3:
File>Project structure> project> set project SDK to 1.7
File>Project structure> SDKs> set JDK homepath to jdk 1.7 path
File> settings> compiler> Java compiler>
Use compiler: Javac
set additional commandline parameters:
-target 7 -bootclasspath C:\Program Files\Java\jdk1.7.0_45\jre\lib\rt.jar
Run> edit configurations> application (and then choose your application) > tick “use alternative JRE” and set path to C:\Program Files\Java\jdk1.7.0_45

Error 4:
Exception in thread "AWT-EventQueue-0" Java.lang.UnsatisfiedLinkError: no lpsolve55j in Java.library.path

Fix 4:
Fixed the error by putting the lpsolve55j.dll and lpsolve55.dll from the readme file into Windows/System32
And leaving out the "VM options" in Run Configurations
(Run>EditConfigurations> leave the textbox in VM options blank)

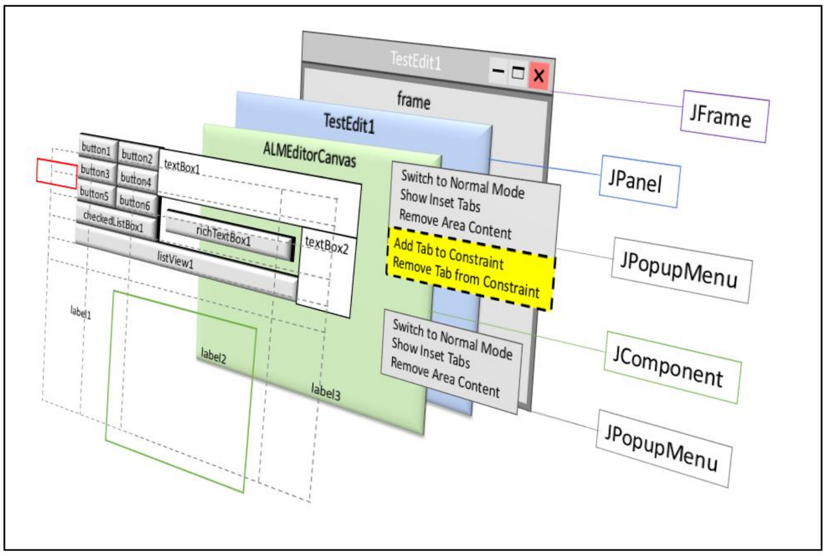
Appendix D - Scala Code Syntax and Concepts Introduction

```
class PropertiesPanel(val parentContainer: java.awt.Container, layout:
ALMLayout) extends Panel {

  /** Example auxillary constructor */
  def this(val parentContainer: java.awt.Container, layout: ALMLayout, tol:
Int){
    this (parentContainer, layout)
    // some code //
  }

  //Member in Panel class
  preferredSize = new Dimension(800,600)

  // Panels for tabbed pane in properties window, for selecting and using the
different editing modes
  var areaPanel: AreaPanel = new AreaPanel(//...//)
  var constraintsPanel: ConstraintsPanel = new ConstraintsPanel(//...//)

  /** Example call to peer to add content*/
  peer.add(new TabbedPane {
    // some code //
    }
    peer.setVisible(true)
  }.peer)

  /** Example method */
  private[editor] def example(flag:Boolean): String = {
    // some code //
    "example return"
  }

}
```

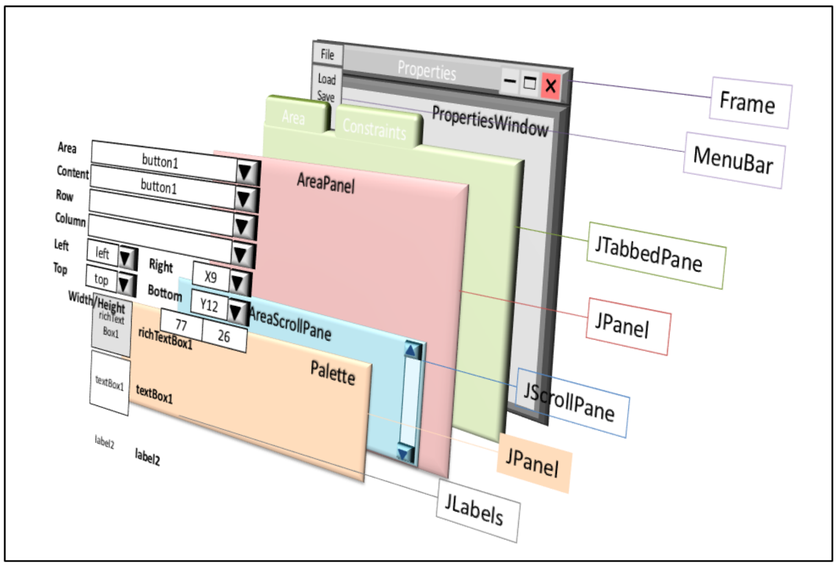
- The code example above illustrates several new concepts in Scala, they are:
- There is no need for a semicolon at the end of each line, it is automatically inferred by the compiler.
- Unlike Java, in Scala the type comes after the variable name, for example `e:ChangeEvent` signifies a parameter named `e` of type `ChangeEvent`.
- In Scala, constants (immutable) are declared with the keyword `val` while mutable variables are declared with the keyword `var`.
- Scala classes have one primary constructor and any number of auxillary classes. The primary constructor is simply the entire class body while any number of auxillary classes can be created by defining a method called `this`. An auxillary constructor must call either an auxillary constructor that has been defined before it, or call the primary constructor. In the example the auxillary constructor allows the passing of one more parameter.
- All constructor parameters become private instance constants (vals) so they can be accessed within methods. To make them public, either put a `var` or `val` declaration in front of them. In the example, `parentContainer` is a public instance constant while `layout` is a private instance constant.
- An example method is provided, all methods uses the `def` keyword and are public by default unless decorated with the `private` keyword. The parameters come after the method name, followed by a colon and the return type. In Scala, the last line of a method will be returned so there is no need for the `return` keyword.

- Scala has more powerful access modifiers than Java. The example method is access-restricted to both the enclosing class and the enclosing editor package.
- The method notation is largely simplified. Calling a method without parameters does not require brackets at the end, for example aLMEditorCanvas.repaint. Furthermore, `+=` and `-=` are also methods denoting addition and removal. In `pages += new Page(…)`, the `+=` method is called by the member pages for adding another page to itself.
- Scala Swing classes provide a number of methods which can be used to specify GUI properties. In the example the preferredSize variable is being assigned.
- In the example `parentContainer` is declared as of type `java.awt.Container` instead of just `Container` as there is also a `scala.swing.Container`. To reduce confusion and to prevent Scala GUI Components from being hidden, the entire `scala.swing` package was imported and Java AWT or Swing classes are only used if they are declared explicitly.
- The `listenTo` method can be passed any object extending the `Publisher` trait. In this example, the `listenTo` method is called in the `tabbedPane` anonymous class with the itself passed as the parameter. The defined reactions will react to any event in fired from the tabbed pane.
- Calling `peer.add` means the `PropertiesPanel` Scala class is treated as its corresponding peer. As a JPanel, it calls the `add` method, which adds the `tabbedPane` as a JPanel.

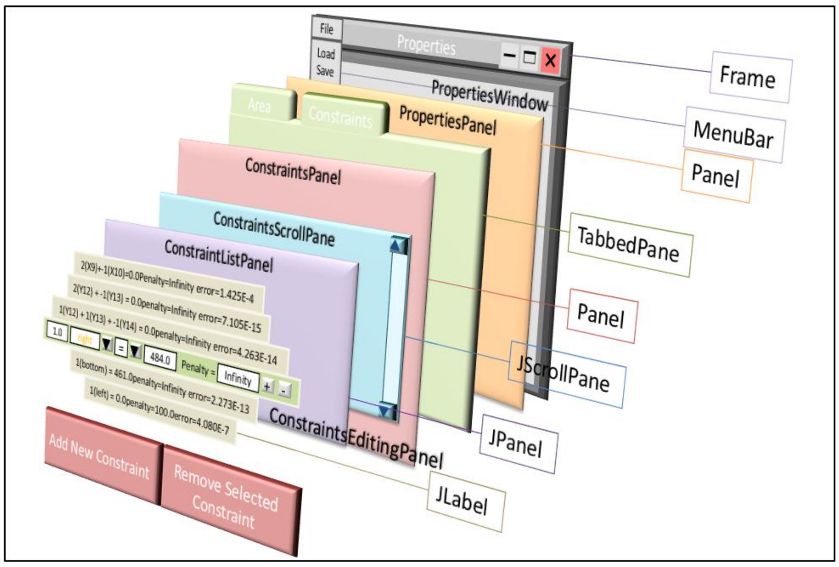
Appendix E - Containment Hierarchy (New Implementation)



Appendix Figure 1 Containment hierarchy for the testing window (new implementation)

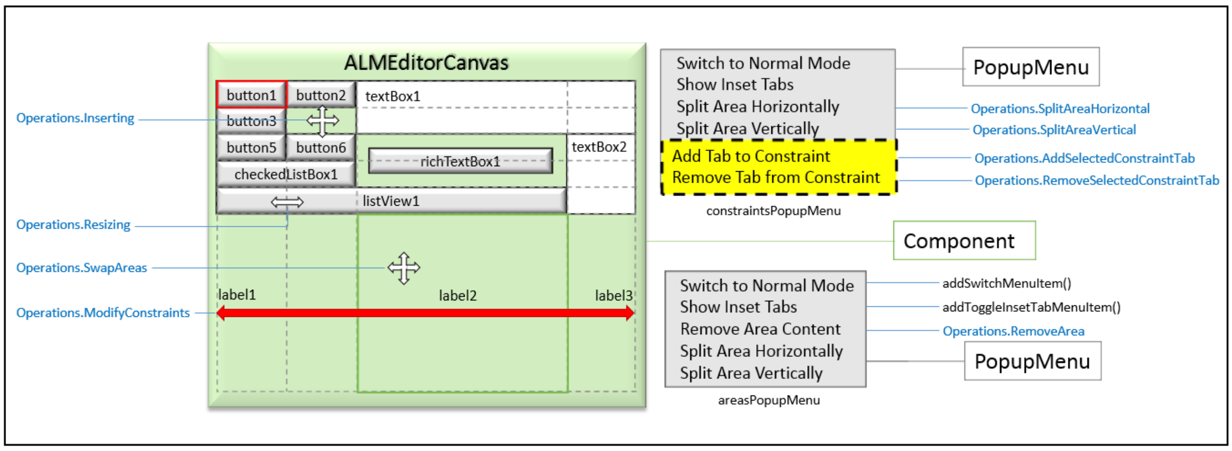


Appendix Figure 2 Containment hierarchy for the properties window (area mode) (new implementation)

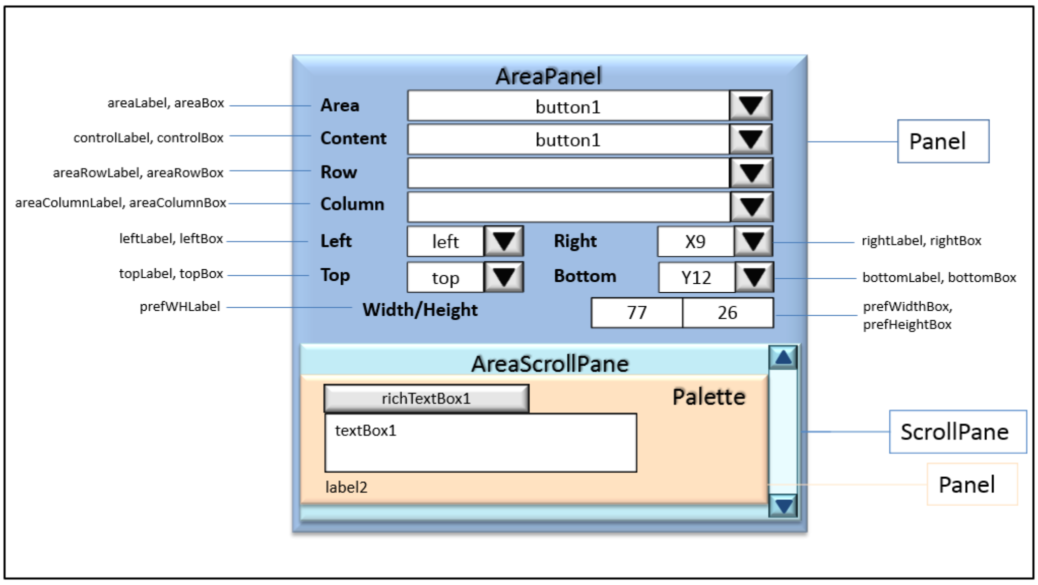


Appendix Figure 3 Containment hierarchy for the properties window (constraints mode) (new implementation)

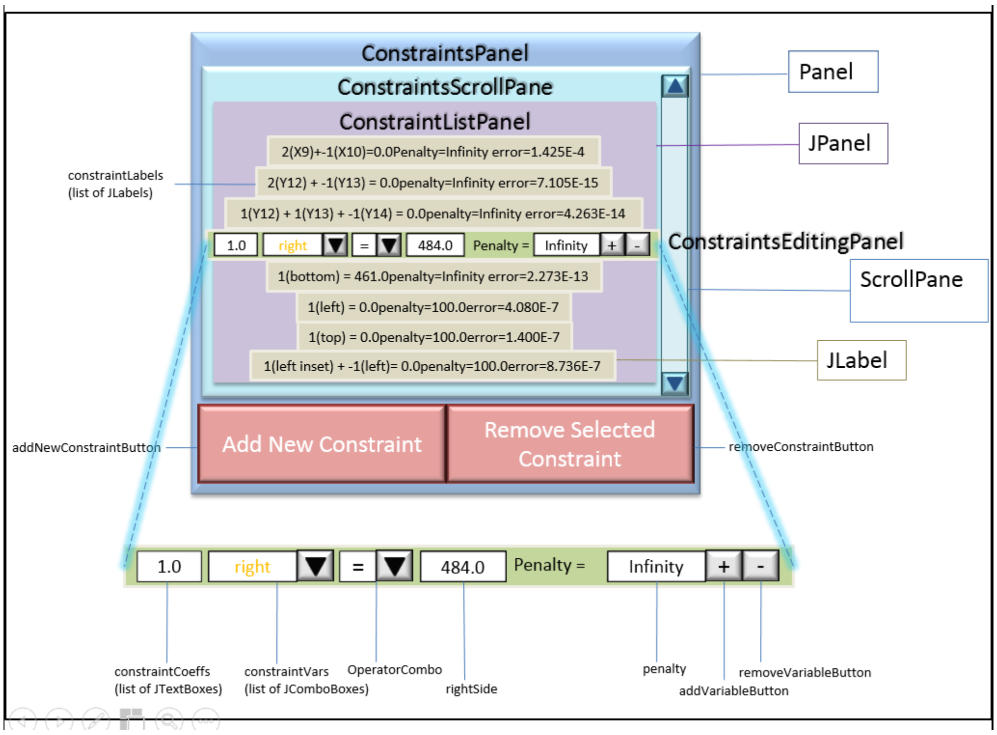
Appendix F - Structure of the Windows (New Implementation)



Appendix Figure 4 Diagram of the testing window after refactoring, containing the variable names of all components in the code (new implementation)

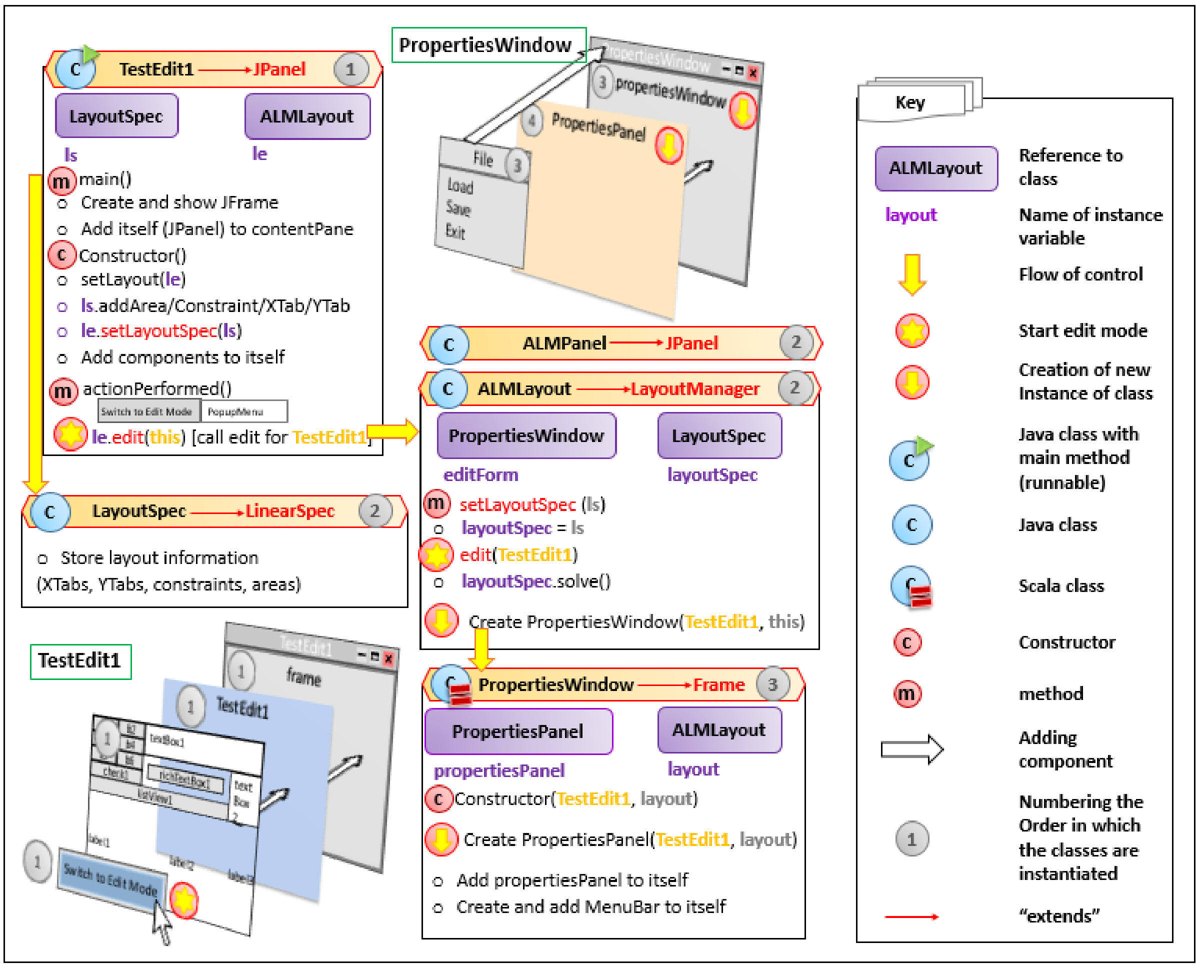


Appendix Figure 5 Diagram of the properties window (area mode) after refactoring, containing the variable names of all components in the code (new implementation)

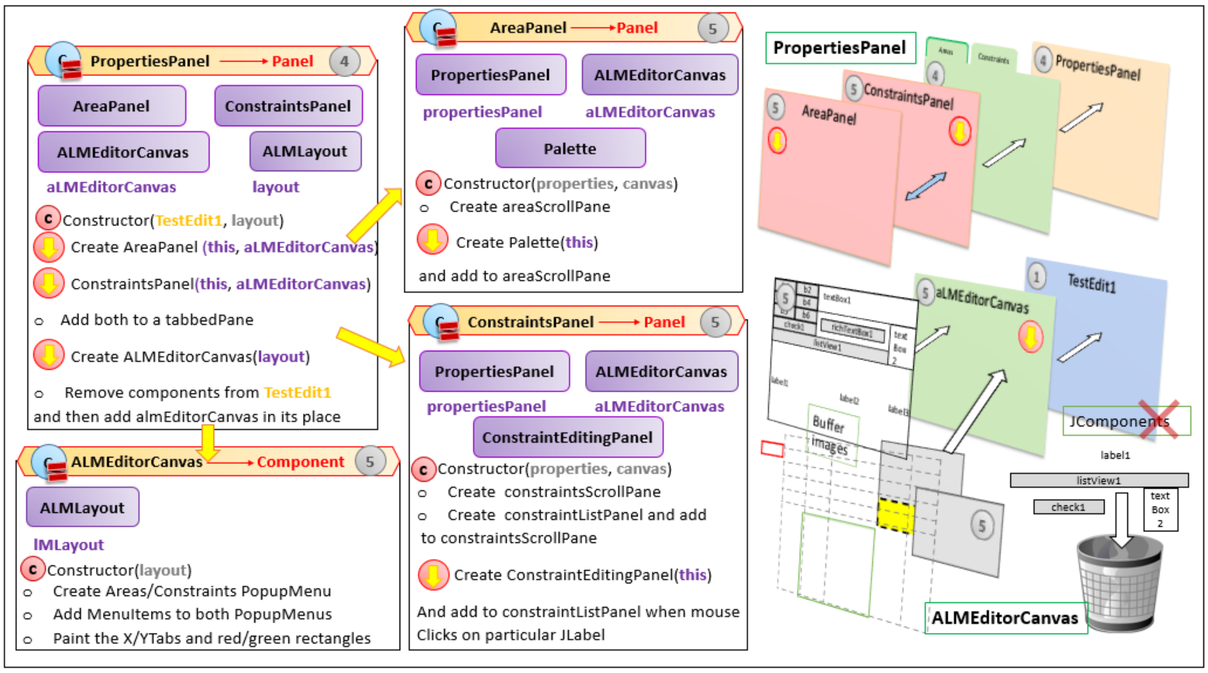

ConstraintsPanel
Panel
ConstraintsScrollPane
ConstraintListPanel
JPanel
2(X9)+-1(X10)=0.0Penalty=Infinity error=1.425E-4
constraintLabels (list of JLabels)
2(Y12) + -1(Y13) = 0.0penalty=Infinity error=7.105E-15
1(Y12) + 1(Y13) + -1(Y14) = 0.0penalty=Infinity error=4.263E-14
1.0
right
=
484.0
Penalty =
Infinity
+
-
ConstraintsEditingPanel
1(bottom) = 461.0penalty=Infinity error=2.273E-13
ScrollPane
1(left) = 0.0penalty=100.0error=4.080E-7
1(top) = 0.0penalty=100.0error=1.400E-7
1(left inset) + -1(left)= 0.0penalty=100.0error=8.736E-7
JLabel
addNewConstraintButton
Add New Constraint
Remove Selected Constraint
removeConstraintButton
1.0
right
=
484.0
Penalty =
Infinity
+
-
constraintCoeffs (list of JTextBoxes)
constraintVars (list of JComboBoxes)
OperatorCombo
rightSide
penalty
removeVariableButton
addVariableButton

Appendix Figure 6 Diagram of the properties window (constraints mode) after refactoring, containing the variable names of all components in the code (new implementation)

Appendix G – Switching a GUI to Editing mode (New Implementation)

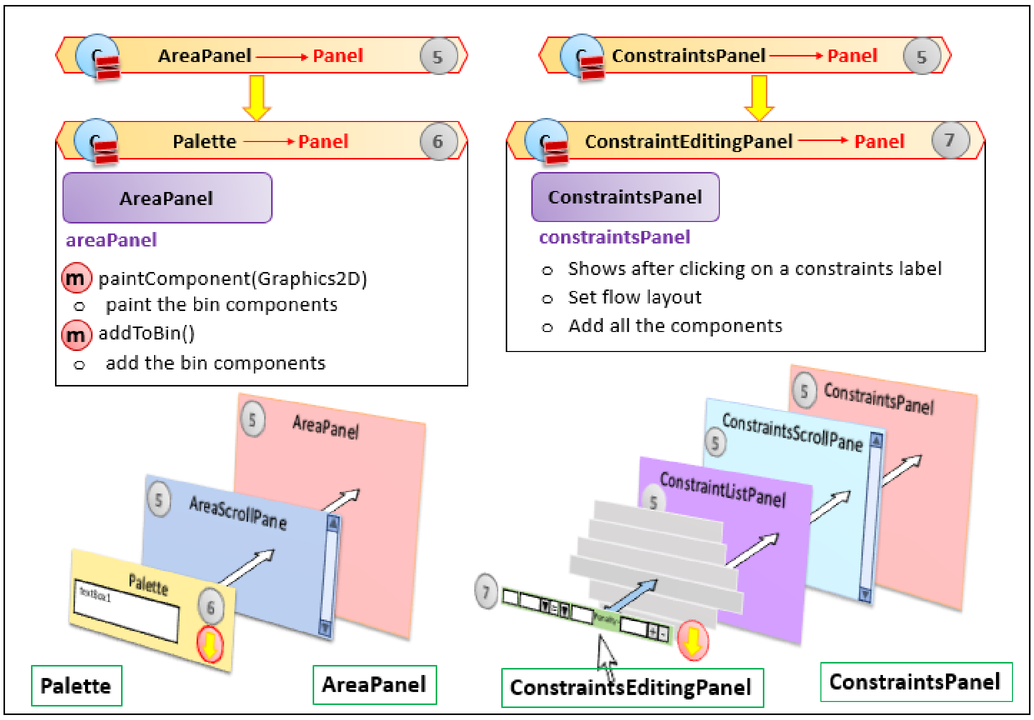


Appendix Figure 7 Showing the initialization of the editing mode by clicking the “Switch to Edit mode” menu item and the instantiation of LayoutSpec, ALMLayout and PropertiesWindow classes. (new implementation)



PropertiesPanel → Panel 4
AreaPanel
ConstraintsPanel
ALMEditorCanvas
ALMLayout
aLMEditorCanvas
layout
Constructor(TestEdit1, layout)
Create AreaPanel (this, aLMEditorCanvas)
ConstraintsPanel(this, aLMEditorCanvas)
Add both to a tabbedPane
Create ALMEditorCanvas(layout)
Remove components from TestEdit1 and then add almEditorCanvas in its place
ALMEditorCanvas → Component 5
ALMLayout
lMLayout
Constructor(layout)
Create Areas/Constraints PopupMenu
Add MenuItems to both PopupMenus
Paint the X/YTabs and red/green rectangles
AreaPanel → Panel 5
PropertiesPanel
ALMEditorCanvas
propertiesPanel
aLMEditorCanvas
Palette
Constructor(properties, canvas)
Create areaScrollPane
Create Palette(this) and add to areaScrollPane
ConstraintsPanel → Panel 5
PropertiesPanel
ALMEditorCanvas
propertiesPanel
aLMEditorCanvas
ConstraintEditingPanel
Constructor(properties, canvas)
Create constraintsScrollPane
Create constraintListPanel and add to constraintsScrollPane
Create ConstraintEditingPanel(this) And add to constraintListPanel when mouse Clicks on particular JLabel
PropertiesPanel
AreaPanel
ConstraintsPanel
PropertiesPanel
aLMEditorCanvas
TestEdit1
Buffer
images
JComponents
label1
listView1
check1
text Box 2
ALMEditorCanvas

Appendix Figure 8 continuing the initialization of the editing mode: the instantiation of PropertiesPanel, ALMEditorCanvas, AreaPanel, ConstraintsPanel classes. (new implementation)



Appendix Figure 9 final of the initialization of the editing mode: the instantiation of Palette and ConstraintEditingPanel classes (new implementation)

Appendix H - Setting up IntelliJ Plugin Development Environment

1. File> New Project> Java > IntelliJ Platform Plugin
2. File > Project structure > platform settings> SDKs> press the "add" button > choose "IntelliJ platform plugin SDK". (its name should be for example IDEA IC-133.331)
3. ClassPath: In the "IntelliJ Platform Plugin SDK home path" select the path to the current installation of the IntelliJ community edition, for example: C:\Program Files (x86)\JetBrains\IntelliJ IDEA Community Edition 13.0.1
4. When the "Select internal Java platform" dialog pops up, select the desired Java SDK version (the recommended version is 1.6 but 1.7 was used and found to be fine for this plugin development).
5. Repeat steps 4 to 8 from Setting up Scala section in Appendix B (this will set up Scala so that the Scala files from the new implementation of ALMEditor can be used)
6. File>Settings> IDE Settings> Ensure PluginDevKit is enabled

Now in the external libraries, should have IDEA IC-133.331 and Scala

To test the plugin:

1) Run>Edit Configurations> Use Class path of module > set it to the current module
2) Press run and a new instance of IntelliJ should start up. Make a new project as normal (only have to do this the first time), setting the project jdk.
3) Make a new java class. The editor tab should automatically show up near the bottom of the window.
4) If it doesn't show, check the edit menu for a menu item called "Put action here..."
5) If it's not there, it means the Actions/Extensions have been registered properly, or IntelliJ is not recognizing them.

Appendix I - Checking out IntelliJ Community Edition

Since IntelliJ IDEA has a open-source UI designer plugin, it could be useful in providing a reference to how GUI editor plugins could be written for IntelliJ. At the time of writing this report, this was the only open-source UI designer plugin. Therefore the community edition source code had to be imported into IntelliJ.

- The latest version of Git was downloaded from: http://git-scm.com/download/win
- Then, the path to Git.exe was set up by selecting: File>Settings>Project Settings>Version Control>Git
- In Path to Git executable, write down the complete path to Git.exe (for example: C:\Program Files (x86)\Git\cmd\git.exe)
- Then, the source code was checked-out by selecting:
  VCS>Check out from version control> Git
- Pasting this URL into Git Repository URL: https://github.com/JetBrains/IntelliJ-community
- Clicking "clone" will start the process of making a copy of this repository.

Appendix J - Errors Encountered During Plugin Development (Section 6.3.4)

Error 1:
Run configuration error: Wrong jdk type for plugin module

Fix 1:
File> Project structure> Project settings> Modules > Dependencies> Set Module SDK to IDEA IC- 133.193

Error 2:
Run Configurations> Use classpath of module is None

Fix 2:
Go to the .iml file and change from <module type="JAVA_MODULE"
to <module type="PLUGIN_MODULE"

Error 3:
Java.lang.IllegalArgumentException: attempt to register provider with non unique editorTypeId: ui-designer

Fix 3:
The above error is caused by the id not being unique
For example in the tool window example in community edition:

```
<extensions defaultExtensionNs="com.IntelliJ">
   <!-- Add your extensions here -->
   <toolWindow id="Calendar"   secondary="true" icon="/general/add.png" anchor="right"
factoryClass="myToolWindow.MyToolWindowFactory"   >

   </toolWindow>h
 </extensions>
```

The toolWindow id "Calendar" is not unique, so produces this error. Have to change the id to something more unique like tools.Calendar

Error 4:
Error message: Could not create the Java virtual machine. when trying to run a plugin (this happens on machines which operate on 32-bit)
See: https://IntelliJ-support.jetbrains.com/entries/23393413

Fix 4:
Go to editConfigurations> VM Options>

there will be some text which says something similar to:
-Xms128m -Xmx550m -XX:MaxPermSize=350m -XX:ReservedCodeCacheSize=96m -XX:+UseCodeCacheFlushing

Reduce the Xmx 100m at a time until the plugin runs

Error 5:
Cannot find path to Git.exe

Fix 5:

This occurs because IntelliJ expects Git.exe for VCS
Download Git.exe from http://git-scm.com/downloads
and go to File>Settings>Version control>Git> and change the path to git executable to point to git.exe

For example:
C:\Program Files\Git\cmd\git.exe

Appendix K - TODOs

- Need to convert the rest of the components to Scala components in the `ConstraintEditingPanel` class
- The areas sometimes start to overlap after some resizing and swapping. The `CheckForOverlap` method in `ALMEditorCanvas` might not be compatible with the changes made in the code during this project
- `binAreaInsets` and `binAreaAlignments` hash maps in `AreaPanel` and `itemsInBin` in `Palette` need to be removed as their function for storing the JComponents, insets and alignment of bin items was replaced by the `ComponentInBin` class.
- Currently the contents and areas are linked in the `AreaPanel` and have the same display name. However the user should be able make an area contain different content
- Within the area panel, the "add new tab" option can be selected in the left, right, top, bottom combo boxes however it is not fully functional
- A very minor problem but in the left, right, top, bottom combo boxes, the different X/YTabs are not displayed as different colours. Within `ConstraintEditingPanel`, the overriding of the method `getListCellRendererComponent` with generic parameters is extremely difficult in Scala. See http://www.Scala-lang.org/old/node/10687
- Concerning the edit operation classes: there isn't a complete separation of concerns since some code are needed by more than one edit operation. Instead of just putting the code in one of the classes, need to put the code within a general reactions registration section in `ALMEditorCanvas`. What if two classes need to assign to the `selectedArea` variable? This code may need to be be in the general `MousePressed` case in `ALMEditorCanvas`.
- Make a more efficient method for updating the widgets in `ConstraintEditingPanel`. Right now, if variables are added or removed to the editing constraint (`addVariableButton` or `removeVariableButton`), `refreshGUIComponents` is called which removes all the GUI components and inserts all of them again. This involves a code duplication; a better way would be to just keep the old components and insert or remove the variables (and updating the list) dynamically.
- There needs to be some way to scroll through the constraint editing panel when the widgets added to it exceeds the amount of space provided by the Panel.
- The load and save functionality in the properties window needs to be tested as they were not considered in this study.