JIST: Java Interaction Separation Toolkit

Adrian-Radu Macocian
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
rmacocian@gmail.com

Dorian Gorgan
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
dorian.gorgan@cs.utcluj.ro

DOI: 10.37789/rochi.2020.1.1.11

ABSTRACT

A GUI toolkit is a library consisting of the elements needed for developing interactive applications. In the 2000s, a lot of effort was devoted to building platforms that enabled the creation of rich internet applications. This let the field of desktop applications underdeveloped. The Java Interaction Separation Toolkit (JIST) was developed with the intention of having a lightweight, cross-platform and support for a declarative UI. By using the WORA aspect of Java most of the desktop platforms are covered. It features XML support for describing user interfaces in a more natural way than the classic wall of text associated with the native code-behind approach.

Author Keywords

Java; GUI sub-system; Graphical User Interface; Interactive Applications; Markup Language; Functional and Interaction Separation; GUI toolkit.

ACM Classification Keywords

H.5. Information interfaces and presentation (e.g., HCI)

INTRODUCTION

Graphical User Interfaces are the main reason why the personal computer reached the mainstream success it has now. In year 1960 the idea of a Graphic User Interface (GUI) started to take shape, and it changed a few times until it reached its peak together with the historical launch of the Window 95 in 1995 [2]. Most current GUI toolkits were created in the late 90s, early 00s. They may be well maintained, but the foundations of the toolkits were created in a time when the environment for graphical interfaces was completely different, and that takes its toll. Computation power is abundant and it's becoming easier to provide the functional requirements of an application. In these circumstances, the choice of software is made based on user friendliness and fluidity of the interface design.

Developing user interfaces is mostly done using a GUI toolkit or framework. Those toolkits handle all the hardware inputs and outputs, define the interaction techniques and provide the developer with the tools for giving input choices to the user, and for handling that input from the user.

This paper will describe the Java Interaction Separation Toolkit (JIST). It is a GUI toolkit developed completely in Java with no external dependencies, that focuses on separating the functional and the interactive components of interactive applications using a markup language. By avoiding any external dependencies, it is ensured that the toolkit may be used on any system that supports the Java Virtual Machine (JVM).

MOTIVATION

It is possible to create a complex user interface using the existing solutions, but it is unnecessary difficult. The current solutions (especially in Java) require nesting of elements using layout managers to ensure that the software will look the same independently of the platform on which it runs. This causes walls of text and makes it so that the code is impossible to be read.

Markup languages can be used to provide layouts to elements in a more natural way and makes visualizing those layouts easier. Windows WPF takes full advantage of this aspect with the XAML description of interfaces [8][9].

There was an attempt of having markup language support in Java with the JavaFX and the FXML (an XML-based language used for describing user interfaces), but JavaFX's future is an uncertainty at this point [7]. Even without the uncertainty surrounding JavaFX's future, using FXML is difficult and seems like an additional feature instead of a core functionality of the system.

The purpose of JIST is to create a platform for describing user interfaces which can separate the aspect from the behavior of the application. This decoupling can enable teams to work concurrently and makes the software easier to understand and maintain. The problem of having software look the same, independently of the platform can be solved by providing context-relative sizes and locations. By specifying everything relative to another element, the developer should always understand how the application should look. This way it is possible to achieve similar displays independent of the system which runs them, without the need of using multiple nested layout managers.

The combination of providing a separation between the functional and the interactive component, providing contextual sizes and location to elements, and the ease of developing layouts in xml is the reason why JIST brings a new approach to the field of GUI toolkits.

OBJECTIVES

The main objective of this paper is having a cross-platform solution for developing Graphical User Interfaces which supports a markup language for the description of the layouts of applications.

Cross-Platform

The platform most often represents the operating system which the application runs on. Covering more than 97% of the market share of desktop OS is done by making sure that the system can be ran in Windows, OS X and Linux [4]. Since all the aforementioned operating systems are able to run a Java Virtual Machine (JVM), developing JIST in pure java should be able to cover the cross-platform objective.

Figure 1 presents the desktop operating system market share as measured by [4].

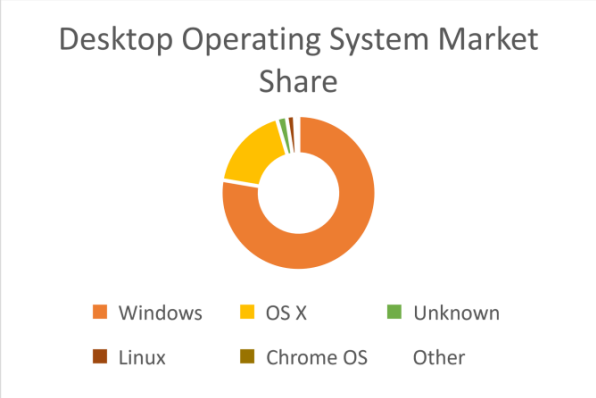

Figure 1. Desktop Operating Systems Market Share [4]

Contextual Size and Location

Using absolute sizes and locations is, rightly so, frowned upon when developing user interfaces. That is because it is impossible to know on what screen the software will be run on, so therefore we can't predict how the software will look like. Current Java native solutions have solved this issue with the use of layout managers. The problem for a complex user application, there will need to be a lot of nested layout managers, which makes it hard to keep track of everything.

The solution used in JIST is one that is already present in the field of web applications, more specifically in HTML. That is, using locations and sizes relative to the element on which the component is displayed. This way, the size of the screen should not influence the overall look of the user interfaces.

Markup Language Support

The separation between the functional and the interactive components of an interactive application could be achieved, in the way JavaFX [3] and WPF [1] also achieved this, by allowing the interface to be described in a markup language. Most GUI toolkits store the elements into a tree-like structure, which is conceptualized more easily in a markup language format.

The support for a markup language was considered from the very beginning of JIST, which ensures that all the components are designed with the goal of supporting markup language in mind. It is important that the markup language feels as a part of the framework, not some feature that may or may not be complete.

The support for the markup language also helps with the problem of having multiple nested layouts. This is due to the nature of markup languages, which allows the visualization tree-like structures in a natural way, as opposed to normal programming languages where it is almost impossible to visualize multiple levels of nested layouts.

ANALISYS

In this section the theoretical foundation on which the project was created will be provided. Here the paper will go a little more in depth into interactive applications, since it is important to know how a tool needs to be used, before designing the tool.

Interaction Applications

An interactive application is composed of two big, and ideally separate, components. The Functional Component, where all the abstract operations on objects are happening and the Interactive Component where the interaction techniques are described together with the interface of objects and operations on those interfaces. The user can only see and act upon the interactive component. The interactive component takes all the input from the user and first validates it, and then processes and transforms it into an application operation that is passed to the functional component.

Interaction techniques are the way in which the user, with the help of the hardware resources and given software components, may provide information to the computer. The results of the interaction are usually visible on screen.

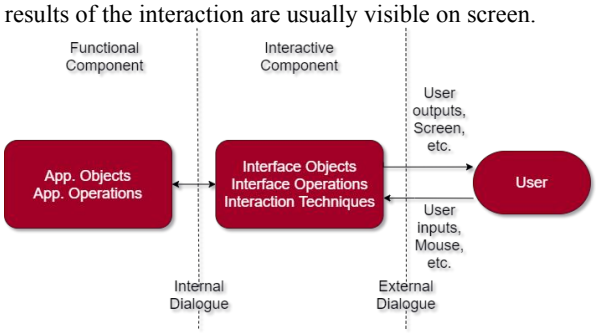

Figure 2. Structure of an Interactive Application [5]

Interaction Techniques

An interaction technique may be informally described as an element which can be graphically represented on screen. Formally, an interaction technique is a way in which the user, with the help of the hardware resources and software components, may provide information to the computer. The interaction technique usually is composed of an input device and an interaction element.

Model of an Interaction Technique

An interaction technique is the way in which a user may communicate with an application with the purpose of achieving a simple action. It may be a simpler way of visualizing the actual communication. Most interaction techniques can be described in the format of an interaction cycle.

The interaction cycle is composed of a prompter, symbol, echo and value. This interaction cycle helps with the better visualization of how a user communicates with the software. An Interaction technique begins with the prompter stage of the cycle, when something is selected or focalized, and the system lets the user know that some form of input is accepted. In the symbol stage, the user will provide some input which will be validated. The echo is the system's way to show some feedback to the user to confirm that the input was received and, finally, in the value stage, the value will be modified to what the application accepts (i.e. normalization).

In most techniques, the user has multiple possible available valid actions (such as clicking, dragging, moving the mouse). An interaction technique can be a metaphor or a symbolical representation of a real operation, which should help us visualize the operations. The metaphor has a visual presentation (some shape or drawing on the screen), a scenario (a way in the user may interact with it), a sequence of user actions (a set of permitted actions) and an interaction device (usually an input device: mouse, keyboard, etc.).

Event Based Control

Most interactive applications are event driven. This means that during most of its lifecycle, the application is waiting for some user events to happen, to which it will respond based on some predefined procedures. The response time to those events must be as low as possible for a satisfying experience for the user.

Figure 3 shows the flow of an event-based control.

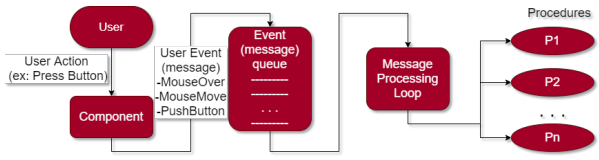

Figure 3. The structure of event-based control [6]

In the event-based control of the application, the user first performs an action which an interaction technique will capture. That action is then transformed into an event that is passed to an event queue. The event queue works as a FIFO 1 list. The queue passes the events, in order, to a message processing loop. The loop decides on which procedure the call for each message. A procedure is a set of actions that are designed to resolve the interaction with the user.

Conceptual Architecture

We will start discussing the conceptual architecture with the most basic objective: Displaying objects on the screen. In order to display something on the screen we need to use OS system calls for creating a new window and then for drawing on that window. A specialized library for hardware interaction will be used. This will also solve the problem of receiving and interpreting user input.

Creating the aspect of the application can be represented as a tree of graphical elements, where the leaves are in the front of the screen and the root in the back of the screen. In order to create such a tree, a common class is needed, which will act as the nodes in the tree. This class should also implement all the methods needed by most of the visual components (such as painting, checking for collision, setting the location and size, etc.). This tree of elements should be passed to the window and then the window will display them on the screen.

It was noted that supporting a markup language for the layouts is a big objective. The markup language should be interpreted at run-time and then a visual tree should be generated following a description in markup language. The choice to use the standard XML notation for the layouts was done due to its flexibility and structure [5].

Besides the visual elements, for modularity, there should be an extra element that deals with the decorations (such as borders and effects). Having them described separately from the main class will give more flexibility in designing applications and for future changes.

With all those choices in mind, the next step is to present a conceptual architecture for the system (Figure 4). Two libraries have been added which are present in the native JDK so that any system that is compatible with Java will be compatible with this system. Those two libraries are: Swing for interacting with the hardware, and XML DOM which is used by the parser.

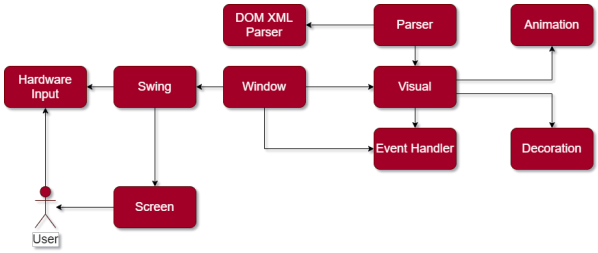

Figure 4. The conceptual architecture of JIST

RELATED WORK

With user interfaces being as big as they currently are, naturally there exist a lot of current frameworks for developing GUIs. It is not practical to try to compare this solution to all of the other available solutions, as they are so numerous. There also isn't a universal best toolkit in this field, and everyone has their own preferences. The following paragraphs will describe two of the most widely used frameworks in Swing and WPF. JavaFX will also be presented as at one point it was supposed to be the successor of Swing.

Swing

Swing was designed with a modified model-view-controller design pattern. It uses the UI component as both the view and the controller. It was developed entirely in JAVA for cross-platform support and easier maintenance. It supports multiple look-and-feels so that it feels native in the platform it runs in. But the look-and-feel of the application may also be changed at runtime.

Swing was developed as an upgrade to the existent AWT API, so it has full compatibility with AWT components. It handles look-and-feel characteristics in a UIManager class, which communicates with each component's UI object to control the display.

JavaFX

It was initially released in 2008 as the successor to Swing, which was supposed to create both web and desktop applications with ease. Since then the web application support has been deprecated and JavaFX started focusing solely on desktop applications. It features its own markup language, the FXML, for declarative description of interfaces. The structure is separated into stages and scenes. Each stage is a window, but it may support multiple scenes, although only one scene is active at a time. All the elements in a scene create a scene graph. The user interface is not native, but it supports Cascading Style Sheets (CSS) for personal touches to applications.

Windows Presentation Foundation (WPF)

WPF: Windows Presentation Foundation is the graphical sub-system developed by .Net Foundation under Microsoft. It was released in open source in December 2018 together with WinForm and WinUI and is the go-to system for developing Graphical User Interfaces using the .Net framework.

All display in WPF is done through DirectX so it relies on Windows for it to function. This also means that it is significantly more efficient in hardware and software rendering. It is usually the go-to platform for developing desktop applications that are only supposed to work on Windows.

WPF values properties a lot higher than events. The goal is for the system to have multiple properties that control the flow of the application. Changes are signaled through notifications. Dependencies are handled automatically, and any property change triggers a dependency revalidation. Any object can provide other objects definitions of its properties.

IMPLEMENTATION

Here the design choices and how most of the framework was implemented will be laid out.

Storing Elements in Memory

The Window class has an instance of a Java Swing JFrame which deals with drawing the final virtual image on the screen. The reason for using Swing is that this ensures the platform has as few dependencies as possible, and the Swing library is included in the native JDK. Besides this, Swing handles all the system calls for hardware interrupts. The window class acts as an interface between this solution and the Swing library.

The Visual class is the backbone of the entire structure. Through this class all the information that should not be accessible to the user is shared, such as the virtual images of the components and handling of user events. With the help of this class, the system may create a visual tree which will later be used for passing graphics information (figure 5). Each node in the tree (which is visible on screen) has a virtual image assigned.

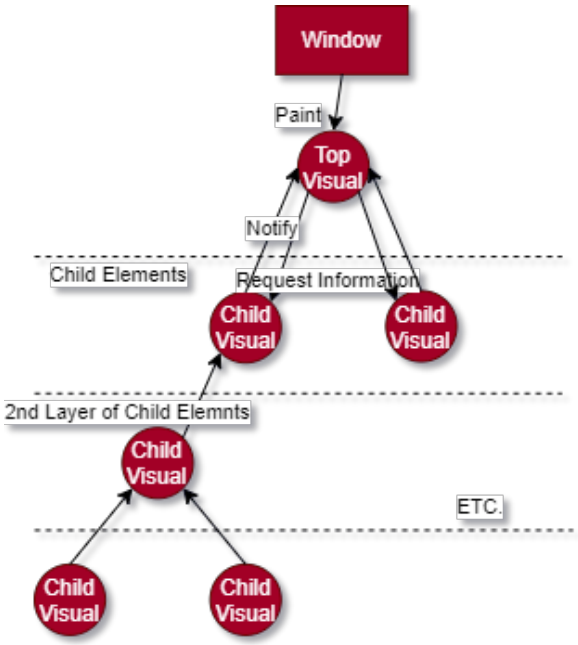

Figure 5. Conceptualization of the visual tree

Any parent node has access to all its children nodes through the findByName method. The parent node decides when and how to place the children nodes in its visual image. A node that sustained a change which requires a repaint needs to signal all the way up the tree that the repaint is necessary. The request is propagated up the tree and only the direct

ancestors of that node will need to be repainted while the other nodes remain valid.

The only way for children to pass information to a parent is through notifications, which the parent decides if and how to handle.

Event Handling

The importance of user events for any graphical user interface cannot be overstated, so triggering them appropriately is a must. There are some rules which describe how events are passed to the components on the screen:

- Only one element on the screen can have the focus
- The mouse is considered on top of an element if the collision method (isInside) for the mouse coordinates returns true
- Mouse events are triggered only on the front-most component (which returned true for isInside)

The framework currently supports three kinds of events: mouse events, keyboard events and mouse wheel events.

The window gets the hardware information from Swing and then passes that information down the visual tree until the right component is reached. The keyboard, mouse pressed, and mouse wheel events are passed directly to the focused element, which is stored as a singleton in the window.

Drawing on the Screen

Each visual component is assigned a virtual image the moment when it is added to a window (so it is displayable). The window class extends the visual class, so it also has a virtual image, which is passed to the JFrame the moment a frame needs to be drawn. This ensures that all the frames drawn on screen are complete images using double buffering.

Every node first applies its own graphic logic on the virtual image and afterwards paints the child nodes' virtual images on top of its own image. This way, if any node in the visual tree needs to be updated, it will only affect the direct ancestor nodes. Any other node can keep painting the same virtual image with no repainting needed.

The moment a new window is created, a *Painting Thread* will also be created. The window also stores the information on the number of frames to be displayed per second. The painting thread makes sure that frames are displayed at the correct rate, and that each image is the most up to date image the system has.

The painting algorithm is composed of two methods: revalidate and repaint. The thread calls the repaint method for every frame. The method first runs all the animations, and then the method checks if any component needs revalidation, if this is not the case, then the virtual image of the Window is still up to date and can be displayed on the screen as is. If a component was changed and the image needs to be updated, then the revalidation method is called. In revalidation, any outdated nodes in the visual tree will clear their virtual images, and then proceed to repaint them to be up to date.

Since when a node requests an update, all the ancestors of that given node need to be updated also, all the updates requests will propagate all the way up to the window. This way, if the window doesn't require any updates, neither does any other node in the visual tree, and the check can be done in O(1).

Window

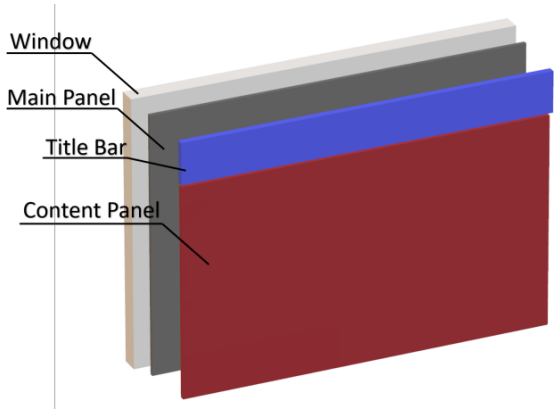

Figure 6. The structure of the window

Any GUI application using JIST requires a Window to function. All the visual elements must be placed inside a visual tree which has a window as the root. This ensures both that the element is displayed on screen, and that the user can interact with that element using the mouse or the keyboard.

The window is first created with a generic title bar. The title bar may be removed or replaced at any time. Any new title bar must extend the title bar class. The active title bar is also an instance of Visual so some attributes such as the colors may be changed as necessary without the need to change the entire title bar.

The window is composed of a Main Panel which stores a Content Panel and a Title Bar (Figure 6). Any component added with the add Visual method to the window, is automatically added to the Content Panel.

Animations

Animations are a way of displaying multiple images in a very short period which gives us the feeling of movement. Small animations can give an, otherwise bland, application a livelier feel.

Since the animations need to change with each frame, triggering the animations is done in the repaint method, triggered by the Painting Thread. This ensures that before each virtual image is validated, each existing animation is executed with one step.

To make sure that developers can implement their animations with as little hustle as possible, an Animation

Interface is created, which specifies the number of frames per second and a step method, which returns false while the animation is running and true when the animation is done. This is done so that the painting thread can remove finished animations from the window's list of animations.

Animations can be added on any node in the visual tree, and they will be recursively passed all the way up to the window, which stores a list of all the running animations from the components displayed in it.

There are currently 3 types of animations that are used by existing components: color, location and size animations.

XML Parser

Using the XML as the declarative markup language, makes it possible for the system to avoid any external dependencies and to keep the system lightweight.

The parser uses reflection to search for all the classes in the current project or .jar executable, and then matches the tags in xml to classes. This makes it possible to just create a new class which will be usable in xml description right away. The only condition for a class to be declarable in xml is that the class must extend, directly or indirectly, the visual class.

The only knowledge that the parser requires is a string to the xml file that is going to be parsed.

The root element from the XML (which usually is the Window) is first instantiated and has its attributes set, the same goes for all the child nodes until the file is covered. After all the instances are created and have had their attributes set, the parser starts returning bottom-up adding all the nodes to their parent nodes.

Hardware Acceleration

Although it was not an initial requirement, it was important to give to the developers the option of enabling hardware acceleration for the drawing. The first step in enabling hardware acceleration in Java is to set the flags in the JVM. The flags must be set before any graphical processing is done, so it is important that hardware acceleration is enabled first in the project if needed. The second step is setting a flag in the visual class which will cause all the virtual images created to be changed from bufferedImage to volatileImage to ensure that the entire advantage of the hardware acceleration is used.

Contextual Size and Location

The location of an element is given by a locationPlacer, which receives the size of the element and of its parent element, and then it decides on where the element should be placed. The placers are created through a factory pattern so that developers can create their own placing logic. As of right now, there are 10 available placers: top-right, top-center, top-left, middle-right, middle-center, middle-left, bottom-right, bottom-center, bottom-left and a general placer. The first 9 placers do exactly what their name suggest.

The general placer has two parameters, a relative position and an absolute position. The relative position is given in the form of two float numbers between 0 and 1 and describes the position inside the parent element. If the relative position is missing, then the placer will use the absolute position for the location of the element.

The size of elements is decided similarly to how the general placer chooses the location of elements.

EXPERIMENTAL EVALUATION

There were three types of testing done for JIST. Performance testing, scalability testing and integration testing. Afterwards, an evaluation for the usability of the system is provided.

Performance Testing

The performance of the system is decided by the rate in which frames may be repainted, while increasing the depth of the visual tree. The test consists in creating a new window of size 1024 x 576 and adding a panel of the same size. Then before every repaint, ask for the panel to be revalidated and save the number of frames displayed on the screen in one second. To avoid erroneous data, the test was repeated 60 times. After that, a new panel of the same size was added as a child to the last panel, thus deepening the visual tree. Now the new panel was asked for revalidation, which would cause both the panels and the window to be revalidated. The same pattern was repeated until a depth of 30 elements in the visual tree was reached.

The entire test was done two times, the first time the system had no hardware acceleration, and the second time hardware acceleration was activated.

The test checks the performance in the case of constant revalidations which is usually seen in games. Static applications don't usually need to revalidate the image before each frame, but even in these circumstances, without the use of hardware acceleration JIST can display over 30 FPS up to a depth of 7 nested elements (Figure 7).

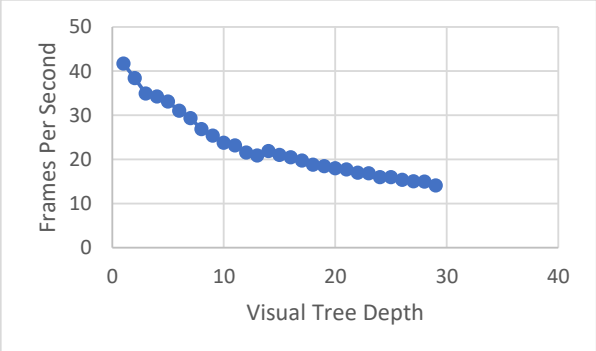

Figure 7. The FPS graph without hardware acceleration

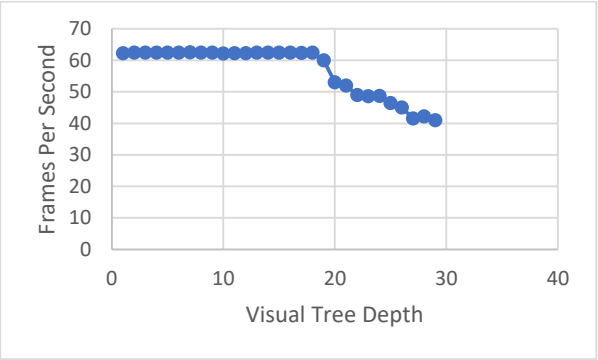

Figure 8. The FPS graph with hardware acceleration

When using hardware acceleration JIST could display 60 FPS, which was the capped value, up until the depth of 17. Even on the depth of 30 the system could display at a rate of 40 FPS.

Scalability Testing

The scalability of JIST may be tested, by checking the number of components that can be added on a visual tree. Again, two cases were considered. The first case was adding elements with a size of 0x0. The test was stopped after 10,000 elements were added, because the system showed no signs of slowing down or troubles.

The second test was done by adding elements of the same size as the window (1024 x 576). This time the creation of elements took a longer time and it seemed as the system would crash. The problem is that each component is given a virtual image of its own size, and the system runs the risk of running out of memory. But in our test case the JVM would always be able to allocate more memory before the system would run out of memory. The test was stopped at 3500 elements, but after 1000 element the creation of new elements started to take considerably longer.

It is worth noting that the elements were all added at the same depth inside the visual tree, to avoid any recursive calls and stack overflow errors. It is very hard to imagine a real case scenario where a user might need more than 3500 elements the size of the screen. The test does show that it takes considerably more time to create new elements the larger they are, and this causes rises in the response time and falls in performance while the system handles the creation of the element. The response time and performance quickly readjust once the elements are created.

Integration

For integration testing, multiple applications were developed. All through the development of JIST new applications were developed with the purpose of seeing how the system handles real scenarios.

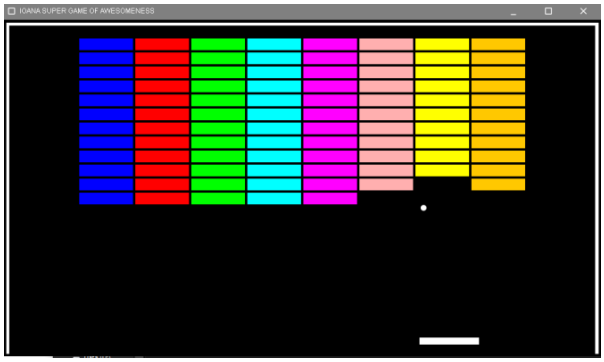

Figure 9. Breakout replica

Figure 10. A mock application

Figure 11. A Chess game

All the above figures (figures 9, 10 and 11) are applications developed completely in JIST. The layouts were written completely in xml.

Usability

It is hard to rate the usability of such a project objectively, since the toolkit choice of each developer is very much subjective.

The usability is described in [6] as: “the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”. The usability is highly related to the target audience of the system. The target

audience is the software developers that want to create desktop graphical user interfaces in Java.

Several efforts were made to make the toolkit as friendly as possible to new developers and to keep it extendable so that everyone may implement their own vision. A few examples of those efforts are making sure that any class is usable in xml description, making sure that every functionality can be extended upon and providing a set of interaction techniques or widgets that are required in almost any interface. The set of available components is still expanding, but currently consists of:

- Buttons and toggle button
- Text boxes and input text boxes
- Panels, scrollable panels and grid panels
- Check boxes and radio buttons
- Dropdown menus
- Sliders
- Images

Another part of usability was giving the developers the possibility to reference images by just specifying the name. The developer can just add an image in .png format to the class-path and reference it through just the name.

CONCLUSIONS

In conclusion, it is entirely possible to develop both static applications and games using JIST. The final library is lightweight and with a set of icons bundled into it, the size does not exceed 200 KB.

What separates JIST from other available solutions is that: it was developed with the intention of describing layouts in a markup language, it is lightweight, and it is easy and straight-forward to use without compromising in the performance, customizability or response time departments.

REFERENCES

1. Anderson, C. "*Essential Windows Presentation Foundation (WPF)*", Addison-Wesley Professional, 2009
2. Barnes, S. B. "User friendly: A short history of the graphical user interface", *Sacred Heart University Review: Vol 16, Issue 1*, Article 4, 2010
3. Clark, J., Connors, J., Bruno, E. "*JavaFX: Developing Rich Internet Applications*", Addison-Wesley Professional, 2009
4. Desktop Operating System Market Share Worldwide, https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201906-202006, visited: 20-jun-2020
5. Harold, E. R. "*Processing XML with JavaTM: A Guide to SAX, DOM, JDOM, JAXP, and TrAX* ", Addison-Wesley Professional, 2002
6. IOS, "*Ergonomics of human-system interaction — part 11: Usability: Definitions and concepts*", 2018
7. Oracle, "*Java Client Roadmap Update*", 2018, https://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf, visited: 17-nov-2019
8. Subhashini, C., Premalatha, S., "XAML - a user interface markup language", i-*manager's Journal on Software Engineering, 4(1)*, pp. 1-3, 2009
9. Macvittie L. A., "XAML in a Nutshell: A desktop Quick Reference (In a Nutshell O'Reilly))", O'Reilly Media Inc, USA, 2006