Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №5

з дисципліни
«Об'єктно орієнтоване програмування»

Виконав:
Студент групи ІМ-21
Сірик Максим Олександрович
номер у списку групи: 22

Перевірив:
Порєв Віктор Миколайович

Київ 2023

Зміст

1 Мета:

Мета роботи – отримати вміння та навички програмувати багатовіконний інтерфейс програми на C++ в об'єктно-орієнтованому стилі.

2 Завдання:

1. Створити у середовищі MS Visual Studio C++ проект Desktop Application з ім'ям Lab5.
2. Написати вихідний текст програми згідно варіанту завдання.
3. Скомпілювати вихідний текст і отримати виконуваний файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Проаналізувати та прокоментувати результати та вихідний текст програми.
6. Оформити звіт.

2.1 Варіант:

1. Для усіх варіантів завдань необхідно дотримуватися вимог та положень, викладених вище у порядку виконання роботи та методичних рекомендаціях.
2. Номер варіанту завдання дорівнює номеру зі списку студентів у журналі. Студенти з парним номером (2, 4, 6, . . .) програмують об'єкт класу MyEditor на основі класичної реалізації Singleton.

$$22 \bmod 2 = 0 \tag{1}$$

3. Усі кольори та стилі геометричних форм – як у попередньої лаб. роботі №4.
4. Запрограмувати вікно таблиці. Для його відкриття та закриття передбачити окремий пункт меню. Вікно таблиці повинно автоматично закриватися при виході з програми.
5. Вікно таблиці – немодальне вікно діалогу. Таблиця повинна бути запрограмована як клас у окремому модулі. Інтерфейс модуля у вигляді оголошення класу таблиці
6. Запрограмувати запис файлу множини об'єктів, що вводяться
7. Оголошення класів для усіх типів об'єктів робити у окремих заголовочних файлах *.h, а визначення функцій членів – у окремих файлах *.cpp. Таким чином, програмний код для усіх наявних типів об'єктів розподілюється по множині окремих модулів.
8. Ієрархія класів та побудова модулів повинні бути зручними для можливостей додавання нових типів об'єктів без переписування коду вже існуючих модулів.
9. У звіті повинна бути схема успадкування класів – діаграма класів. Побудувати діаграму класів засобами Visual Studio C++.
11. Бонуси-заохочення, які можуть суттєво підвищити оцінку лабораторної роботи. Оцінка підвищується за виконання кожного пункту, з наведених нижче:
1). Якщо у вікні таблиці буде передбачено, щоб користувач міг виділити курсором рядок таблиці і відповідний об'єкт буде якось виділятися на зображенні у головному вікні.
2). Якщо у вікні таблиці користувач може виділити курсором рядок таблиці і відповідний об'єкт буде вилучено з масиву об'єктів. 89
При виконанні бонусів 1 та 2 забороняється робити для цього нові залежності модуля my_table від інших .cpp файлів. Тоді як надіслати повідомлення (наприклад, про виділення користувачем якогось рядка таблиці) від вікна таблиці клієнту цього вікна (наприклад, коду головного файлу .cpp)? Підказки можна знайти у матеріалі лекції стосовно технології Callback, а також патернів Observer, Listener.
3). Якщо програма не тільки записує у файл опис множини об'єктів, а ще й здатна завантажити такий файл і відобразити відповідні об'єкти у головному вікні та вікні таблиці

3 Текст програми:

3.1 Module: com.github.erotourtes.drawing

Лістинг 1: CanvasController.kt

```
package com.github.erotourtes.drawing

import javafx.beans.property.SimpleBooleanProperty
import javafx.beans.property.SimpleObjectProperty
import javafx.scene.canvas.Canvas
import javafx.scene.image.WritableImage
import javafx.scene.paint.Color
import tornadofx.*

class CanvasController(private val canvas: Canvas) {
    fun getSnapshotImage(): WritableImage =
        try {
            WritableImage(canvas.width.toInt(), canvas.height.toInt()).let { image ->
                canvas.snapshot(null, image)
                image
            }
        } catch (e: Exception) {
            WritableImage(1, 1)
        }

    val fillColorProp = SimpleObjectProperty(Color.BLACK)
    val strokeColorProp = SimpleObjectProperty(Color.BLACK)

    fun changeFillColor(color: Color) {
        canvas.graphicsContext2D.fill = color
        fillColorProp.value = color
        gcUpdate.value = updater.not()
    }

    fun changeStrokeColor(color: Color) {
        canvas.graphicsContext2D.stroke = color
        strokeColorProp.value = color
        gcUpdate.value = updater.not()
    }

    fun changeStrokeWidth(width: Double) {
        canvas.graphicsContext2D.lineWidth = width
        gcUpdate.value = updater.not()
    }

    val gcUpdate = SimpleBooleanProperty(false)
    private val updater by gcUpdate
}
```

Лістинг 2: EditorHandler.kt

```
package com.github.erotourtes.drawing

import com.github.erotourtes.drawing.editor.Editor
import com.github.erotourtes.drawing.editor.EmptyEditor
import com.github.erotourtes.drawing.shape.Shape
import javafx.beans.property.SimpleObjectProperty
import javafx.beans.value.ChangeListener
import tornadofx.*

class EditorHandler(
    private val shapeMap: Map<Class<out Shape>, Editor>
) {
    private var curEditor: Editor = EmptyEditor
        set(value) {
            field.disableEvents()
            field = value
            value.listenToEvents()
        }

    private var curShape = SimpleObjectProperty<Shape>()

    val shape: Shape by curShape
    val editor: Editor get() = curEditor

    fun use(pair: Pair<Class<out Shape>, Editor>) {
        with(pair) {
            curEditor = second.apply { shape = getShape(first) }
            curShape.value = curEditor.shape
        }
    }

    fun requestRedraw() = curEditor.redraw()

    fun listenToChanges(subscriber: ChangeListener<Shape>) = curShape.addListener(subscribe

    fun isCurShapeActive(pair: Pair<Class<out Shape>, Editor>): Boolean = pair.first == cu

    private fun getShape(clazz: Class<out Shape>) =
        if (clazz.constructors.isEmpty()) clazz.kotlin.objectInstance!!
        else clazz.getConstructor().newInstance()
}
```

Лістинг 3: GCState.kt

```
package com.github.erotourtes.drawing

import com.github.erotourtes.utils.GCStateSerializer
import javafx.scene.canvas.GraphicsContext
import javafx.scene.paint.Color
import kotlinx.serialization.Serializable

@Serializable
data class GCState(
    val fill: Color,
    val stroke: Color,
    val lineWidth: Double,
) {
    fun apply(gc: GraphicsContext) {
        val s = this
        gc.apply {
            fill = s.fill
            stroke = s.stroke
            lineWidth = s.lineWidth
        }
    }

    fun copy() = GCState(fill, stroke, lineWidth)

    companion object {
        fun from(gc: GraphicsContext) = GCState(
            gc.fill as Color,
            gc.stroke as Color,
            gc.lineWidth,
        )

        val default
            get() = GCState(
                Color.BLACK,
                Color.BLACK,
                1.0,
            )

        val serializer get() = GCStateSerializer
    }
}
```

3.2 Module: com.github.erotourtes.drawing.shape

Листинг 4: Shape.kt

```
package com.github.erotourtes.drawing.shape

import com.github.erotourtes.drawing.GCState
import com.github.erotourtes.utils.Dimension
import com.github.erotourtes.utils.drawOnce
import javafx.scene.canvas.GraphicsContext
import tornadofx.*

abstract class Shape {
    protected var state = ShapeState(
        this::class.java.name,
        Dimension(),
        GCState.default,
    )

    /**
     * Draw the shape without saving the dimension with the default GraphicsContext propert
     */
    abstract fun draw(gc: GraphicsContext, dm: Dimension)

    open fun drawWithState(gc: GraphicsContext) {
        gc.drawOnce {
            state.gcState.apply(this)
            draw(gc, state.dm)
        }
    }

    fun setDm(dm: Dimension) = dm.copyTo(state.dm).let { this }

    open fun copy(): Shape {
        try {
            val shape = this::class.java.getConstructor().newInstance()
            shape.state = state.copy()
            return shape
        } catch (e: Exception) {
            throw RuntimeException("Can't␣copy␣a␣shape")
        }
    }

    open fun getBounds(dm: Dimension) = dm
    fun getBounds() = getBounds(copyDm)

    val copyDm get() = state.dm.copy()
    val copyState get() = state.copy()

    fun setStateWith(state: ShapeState) {
        this.state = state
    }
    fun setGCStateWith(gc: GraphicsContext) {
        state.gcState = GCState.from(gc)
    }

    val x1Prop get() = state.dm.getX1Prop
    val y1Prop get() = state.dm.getY1Prop
    val x2Prop get() = state.dm.getX2Prop
    val y2Prop get() = state.dm.getY2Prop

    class ShapeModel(shape: Shape = EmptyShape) : ItemViewModel<Shape>(shape) {
        val x1 = bind(Shape::x1Prop)
        val y1 = bind(Shape::y1Prop)
        val x2 = bind(Shape::x2Prop)
        val y2 = bind(Shape::y2Prop)

        val isEmptyShape = booleanBinding(itemProperty) {
            item == EmptyShape
        }
    }
}
```

Лістинг 5: ShapeState.kt

```
package com.github.erotourtes.drawing.shape

import com.github.erotourtes.drawing.GCState
import com.github.erotourtes.utils.Dimension
import com.github.erotourtes.utils.ShapeStateSerializer
import kotlinx.serialization.Serializable
```

```
@Serializable
data class ShapeState(
    val clazz: String,
    val dm: Dimension,
    var gcState: GCState,
) {
    fun copy() = ShapeState(clazz, dm.copy(), gcState.copy())

    fun toShape(): Shape {
        val shape = Class.forName(clazz).getConstructor().newInstance() as Shape
        shape.setStateWith(this)
        return shape
    }

    companion object {
        val serializer get() = ShapeStateSerializer
    }
}
```

Лістинг 6: Shapes.kt

```
package com.github.erotourtes.drawing.shape

import com.github.erotourtes.utils.*
import javafx.scene.canvas.GraphicsContext
import javafx.scene.paint.Color
import kotlin.math.abs

class Point : Shape() {
    override fun draw(gc: GraphicsContext, dm: Dimension) {
        val radius = 12.0
        gc.apply {
            val (x, y) = dm.getRaw().second
            fillOval(x, y, radius, radius)
            strokeOval(x, y, radius, radius)
        }
    }

    override fun getBounds(dm: Dimension): Dimension {
        val radius = 12.0
        val (x, y) = dm.getRaw().second
        return Dimension.from(x, y, x + radius, y + radius)
    }
}

class Line : Shape() {
    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply { strokeLine(dm) }
    }
}


class Rect : Shape() {
    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply {
            fillRect(dm)
            strokeRect(dm)
        }
    }
}

class Ellipse : Shape() {
    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply {
            fillOval(dm)
            strokeOval(dm)
        }
    }
}

class Dumbbell : Shape() {
    private val line = Line()
    private val ellipse = Ellipse()

    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply {
            val radius = 24.0
            val hr = radius / 2
            val (start, end) = dm.getRaw()

            line.draw(gc, dm)

            with(ellipse) {
                val d = Dimension

                draw(gc, d.from(start.x - hr, start.y - hr, start.x + hr, start.y + hr))
                draw(gc, d.from(end.x - hr, end.y - hr, end.x + hr, end.y + hr))
            }
        }
    }
}

class CubeEx : Shape() {
    // failed math but got a nice effect
    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply {
            val (s, bg, size) = getCoords(dm)
            val (bgX, bgY) = bg

            strokeRect(s.x, s.y, size, size)
            strokeRect(bgX, bgY, size, size)

            strokeLine(s.x, s.y, bgX, bgY)
            strokeLine(s.x, s.y + size, bgX, bgY + size)
            strokeLine(s.x + size, s.y, bgX + size, bgY)
            strokeLine(s.x + size, s.y + size, bgX + size, bgY + size)
        }
    }
    override fun drawWithState(gc: GraphicsContext) {
        gc.drawOnce {
            state.gcState.apply(this)
            gc.fill = Color.TRANSPARENT
            draw(gc, state.dm)
        }
    }

    override fun getBounds(dm: Dimension): Dimension {
        val (s, bg, size) = getCoords(dm)
        val (bgX, bgY) = bg

        val d = Dimension
        return d.from(s.x, s.y, bgX + size, bgY + size)
    }

    private fun getCoords(dm: Dimension): Triple<Dimension.Point, Pair<Double, Double>, Do
        val (s, e) = dm.getRaw()
        val w = e.x - s.x
        val h = e.y - s.y

        val depthFactor = 0.5
        val size = abs(w.coerceAtLeast(h))
        val depthX = w * depthFactor
        val depthY = h * depthFactor

        val bgX = s.x + depthX
        val bgY = s.y - depthY

        return Triple(s, Pair(bgX, bgY), size)
    }
}

class Cube : Shape() {
    private val square = Rect()
    private val line = Line()

    override fun draw(gc: GraphicsContext, dm: Dimension) {
        gc.apply {
            val (s, bg, size) = getCoords(dm)
            val (bgX, bgY) = bg
            val (sizeX, sizeY) = size

            val d = Dimension

            with(square) {
                draw(gc, d.from(s.x, s.y, s.x + sizeX, s.y + sizeY))
                draw(gc, d.from(bgX, bgY, bgX + sizeX, bgY + sizeY))
            }

            with(line) {
                draw(gc, d.from(s.x, s.y, bgX, bgY))
                draw(gc, d.from(s.x + sizeX, s.y, bgX + sizeX, bgY))
                draw(gc, d.from(s.x, s.y + sizeY, bgX, bgY + sizeY))
                draw(gc, d.from(s.x + sizeX, s.y + sizeY, bgX + sizeX, bgY + sizeY))
            }
        }
    }

    override fun drawWithState(gc: GraphicsContext) {
        gc.drawOnce {
            state.gcState.apply(this)
            gc.fill = Color.TRANSPARENT
            draw(gc, state.dm)
        }
    }

    override fun getBounds(dm: Dimension): Dimension {
        val (s, bg, size) = getCoords(dm)
        val (bgX, bgY) = bg
        val (sizeX, sizeY) = size

        val sX = s.x.coerceAtMost(bgX)
        val sY = s.y.coerceAtMost(bgY)
        val eX = (s.x + sizeX).coerceAtLeast(bgX + sizeX)
        val eY = (s.y + sizeY).coerceAtLeast(bgY + sizeY)

        val d = Dimension
        return d.from(sX, sY, eX, eY)
    }

    private fun getCoords(dm: Dimension): Triple<Dimension.Point, Pair<Double, Double>, Pa
        var (s, e) = dm.getRaw()
        val w = e.x - s.x
        val h = e.y - s.y

        val depthFactor = 0.5
        val sizeX = abs(w)
        val sizeY = abs(h)
        val depthX = w * depthFactor
        val depthY = h * depthFactor

        s = dm.getBoundaries().first

        val bgX = s.x + depthX
        val bgY = s.y - depthY

        return Triple(s, Pair(bgX, bgY), Pair(sizeX, sizeY))
    }
}

object EmptyShape : Shape() {
    override fun draw(gc: GraphicsContext, dm: Dimension) {}
}
```

3.3 Module: com.github.erotourtes.drawing.history

Лістинг 7: Command.kt

```
package com.github.erotourtes.drawing.history

interface Command {
    fun execute()
    fun undo()
}
```

Лістинг 8: Commands.kt

```
package com.github.erotourtes.drawing.history

import com.github.erotourtes.drawing.editor.ShapesList
import com.github.erotourtes.drawing.shape.Shape
import com.github.erotourtes.utils.Dimension

/**
 *  Command that is used to change the whole list of shapes.
 */
class OpenCommand(private val old: ShapesList, private val new: List<Shape>) : Command {
    private val oldList = old.getList().toList()
    override fun execute() {
        old.clear()
        old.addAll(new)
    }

    override fun undo() {
        old.clear()
        old.addAll(oldList)
    }
}

/**
 *  Command that is used to erase the whole list of shapes.
 */
class NewCommand(private val sl: ShapesList) : Command {
    private val copyList = sl.getList().toList()
    override fun execute() {
        sl.clear()
    }

    override fun undo() {
        sl.clear()
        sl.addAll(copyList)
    }
}

/**
 *  Command that is used to add a shape to the list of shapes.
 */
class AddItemCommand(private val shapes: ShapesList, shape: Shape) : Command {
    val shape = shape.copy()
    override fun execute() = shapes.add(shape)
    override fun undo() = shapes.remove(shape)
}
```

```
/**
 *  Command that is used to remove a shape from the list of shapes.
 */
class RemoveItemCommand(private val shapes: ShapesList, private val shape: Shape) : Comman
    override fun execute() = shapes.remove(shape)
    override fun undo() = shapes.add(shape)
}

/**
 *  Command that is used to change the coordinates of a shape.
 */
class ChangeCoordinatesCommand(shapeModel: Shape.ShapeModel, private val redraw: () -> Uni
    private val shape = shapeModel.item
    private val dmNew =
        Dimension.from(shapeModel.x1.value, shapeModel.y1.value, shapeModel.x2.value, shape
    private val dmOld = shapeModel.item.copyDm


    override fun execute() {
        shape.setDm(dmNew)
        redraw()
    }

    override fun undo() {
        shape.setDm(dmOld)
        redraw()
    }
}
```

Лістинг 9: History.kt

```
package com.github.erotourtes.drawing.history

class History {
    private val undoStack = mutableListOf<Command>()
    private val redoStack = mutableListOf<Command>()

    fun undo() {
        undoStack.removeLastOrNull()?.let {
            it.undo()
            redoStack.add(it)
        }
    }

    fun redo() {
        redoStack.removeLastOrNull()?.let {
            it.execute()
            undoStack.add(it)
        }
    }

    fun add(command: Command) {
        undoStack.add(command)
        redoStack.clear()
    }
}
```

3.4 Module: com.github.erotourtes.drawing.editor

Лістинг 10: DmProcessor.kt

```kotlin
package com.github.erotourtes.drawing.editor

import com.github.erotourtes.utils.Dimension

fun interface DmProcessor {
    fun process(dm: Dimension): Dimension
}
```

Лістинг 11: Editor.kt

```kotlin
package com.github.erotourtes.drawing.editor

import com.github.erotourtes.drawing.history.AddItemCommand
import com.github.erotourtes.drawing.history.History
import com.github.erotourtes.drawing.shape.*
import javafx.scene.paint.Color
import com.github.erotourtes.utils.*
import javafx.collections.ListChangeListener
import javafx.scene.canvas.GraphicsContext
import javafx.scene.input.KeyEvent
import javafx.scene.input.MouseEvent

abstract class Editor {
    protected lateinit var shapes: ShapesList
    protected lateinit var gc: GraphicsContext
    protected lateinit var history: History
    private var _shape: Shape = EmptyShape

    var shape: Shape
        get() = _shape
        set(value) {
            this._shape = value
        }

    fun init(shapes: ShapesList, gc: GraphicsContext, history: History, shape: Shape = Emp
        this.shapes = shapes
        this.gc = gc
        this.history = history
        this.shape = shape
    }

    protected val dm = Dimension()

    protected open var curProcessor: DmProcessor = DmProcessor { it }
    protected open val processor: DmProcessor = DmProcessor { it }
    protected open val altProcessor: DmProcessor = DmProcessor { Dimension.toCorner(it) }
    protected open val ctrlProcessor: DmProcessor = DmProcessor { Dimension.toEqual(it) }

    protected open val shapesChangeListener = ListChangeListener<Shape> { redraw() }
    protected var isStillDrawing = false

    open fun listenToEvents() {
        val c = gc.canvas
        c.isFocusTraversable = true
        c.requestFocus()
        c.setOnMousePressed(::onMousePressed)
        c.setOnMouseDragged(::onMouseDragged)
        c.setOnMouseReleased(::onMouseReleased)
        c.setOnKeyPressed(::onKeyPressed)
        c.setOnKeyReleased(::onKeyReleased)
        shapes.addListener(shapesChangeListener)
    }

    open fun disableEvents() {
        with(gc.canvas) {
            onMousePressed = null
            onMouseDragged = null
            onMouseReleased = null
            onKeyPressed = null
            onKeyReleased = null
        }
        shapes.removeListener(shapesChangeListener)
    }

    protected open fun onMousePressed(e: MouseEvent) {
        gc.canvas.requestFocus()
        redraw()
        dm.setStart(e.x, e.y)
        isStillDrawing = true
    }

    protected open fun onMouseDragged(e: MouseEvent) {
        redraw()

        dm.setEnd(e.x, e.y)
        previewLine()
    }

    protected open fun onMouseReleased(e: MouseEvent) {
        isStillDrawing = false
        if (e.isDragDetect) return // returns if mouse was not dragged
        shape.setDm(curProcessor.process(dm))

        shape.setGCStateWith(gc)
        val command = AddItemCommand(shapes, shape).also { it.execute() }
        history.add(command)

        redraw()
    }

    protected open fun onKeyPressed(e: KeyEvent) {
        changeProcessor(e)
```

```
        if (!isStillDrawing) return
        redraw()
        previewLine()
    }

    protected open fun onKeyReleased(e: KeyEvent) = onKeyPressed(e)

    protected open fun changeProcessor(e: KeyEvent) {
        curProcessor = processor
        if (e.isControlDown) curProcessor = pipe(curProcessor, ctrlProcessor)
        if (e.isAltDown) curProcessor = pipe(curProcessor, altProcessor)
    }


    private fun drawAll() {
        for (shape in shapes) shape.drawWithState(gc)
    }

    private fun clear() = gc.clearRect(0.0, 0.0, gc.canvas.width, gc.canvas.height)

    fun redraw() {
        clear()
        drawAll()
    }

    protected open fun previewLine() {
        gc.drawOnce {
            setPreviewProperties()
            shape.draw(this, curProcessor.process(dm))
        }
    }

    protected fun setPreviewProperties() {
        gc.setLineDashes(5.0)
        gc.fill = Color.TRANSPARENT
    }

    fun highlight(shape: Shape) {
        redraw()
        gc.drawOnce {
            setHighlightProperties()
            val shapeArea = shape.getBounds().apply {
                val (first, second) = getRaw()
                val nX = if (second.x > first.x) 1 else -1
                val nY = if (second.y > first.y) 1 else -1
                val size = 15
                setStart(first.x - size * nX, first.y - size * nY)
                setEnd(second.x + size * nX, second.y + size * nY)
            }

            Rect().draw(gc, shapeArea)
        }
    }
    protected fun setHighlightProperties() {
        gc.setLineDashes(5.0)
        gc.lineWidth = 5.0
        gc.stroke = Color.BLUEVIOLET
        gc.fill = Color.TRANSPARENT
    }
}
```

Листинг 12: Editors.kt

```
package com.github.erotourtes.drawing.editor

import com.github.erotourtes.drawing.history.AddItemCommand
import com.github.erotourtes.utils.SingletonHolder
import javafx.scene.input.MouseEvent

class PointEditor private constructor() : Editor() {
    override fun onMousePressed(e: MouseEvent) {
        shape.setDm(dm.setEnd(e.x, e.y))
        redraw()
        previewLine()
    }

    override fun onMouseDragged(e: MouseEvent) {
        onMousePressed(e)
    }

    override fun onMouseReleased(e: MouseEvent) {
        shape.setGCStateWith(gc)
        val command = AddItemCommand(shapes, shape).also { it.execute() }
        history.add(command)

        redraw()
    }

    companion object : SingletonHolder<PointEditor>({ PointEditor() })
}

object EmptyEditor : Editor() {
    override fun listenToEvents() {
        disableEvents()
    }
}

object ShapeEditor : Editor()
```

Листинг 13: ShapesList.kt

```
package com.github.erotourtes.drawing.editor

import com.github.erotourtes.drawing.shape.Shape
import com.github.erotourtes.drawing.shape.ShapeState
import javafx.collections.FXCollections
import javafx.collections.ListChangeListener
import javafx.collections.ObservableList
```

```
class ShapesList : Iterable<Shape> {
    private val shapeArr = FXCollections.observableArrayList<Shape>()

    val size: Int
        get() = shapeArr.size

    fun add(sh: Shape) {
        shapeArr.add(sh)
    }

    fun addAll(shapes: List<Shape>) {
        shapeArr.addAll(shapes)
    }

    fun addListener(listener: ListChangeListener<Shape>) = shapeArr.addListener(listener)

    fun removeListener(listener: ListChangeListener<Shape>) = shapeArr.removeListener(liste

    fun remove(sh: Shape) {
        shapeArr.remove(sh)
    }

    fun clear() = shapeArr.clear()

    fun getList(): List<Shape> = shapeArr.toList()

    fun getStatesList(): List<ShapeState> = shapeArr.map { it.copyState }

    fun getObservableList(): ObservableList<Shape> = shapeArr

    override fun iterator(): Iterator<Shape> = shapeArr.iterator()
    override fun toString(): String = "ShapesList(${shapeArr.size})"
}
```

3.5 Module: com.github.erotourtes.view

Лістинг 14: MainController.kt

```
package com.github.erotourtes.view

import com.github.erotourtes.drawing.CanvasController
import com.github.erotourtes.drawing.EditorHandler
import com.github.erotourtes.drawing.history.History
import com.github.erotourtes.drawing.editor.*
import com.github.erotourtes.drawing.shape.*
import com.github.erotourtes.utils.EditorInfo
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIcon
import javafx.scene.canvas.Canvas
import javafx.scene.layout.Pane
import tornadofx.*

class MainController : Controller() {
    private val cm by inject<CanvasModel>()
    private val eim by inject<EditorsInfoModel>()
    private val canvas = Canvas()
    private val shapeList = ShapesList()
    private val history = History()
    private lateinit var editorHandler: EditorHandler

    private val editorsInfo = listOf(
        EditorInfo("Dot", "Dot", Point::class.java to PointEditor.getInstance(), FontAweso
        EditorInfo("Line", "Line", Line::class.java to ShapeEditor, FontAwesomeIcon.MINUS)
        EditorInfo("Rectangle", "Rectangle", Rect::class.java to ShapeEditor, FontAwesomeI
        EditorInfo("Ellipse", "Ellipse", Ellipse::class.java to ShapeEditor, FontAwesomeIc
        EditorInfo("Dumbbell", "Dumbbell", Dumbbell::class.java to ShapeEditor, FontAwesom
        EditorInfo("Cube", "Cube", Cube::class.java to ShapeEditor, FontAwesomeIcon.CUBE),
        EditorInfo("CubeEx", "CubeEx", CubeEx::class.java to ShapeEditor, FontAwesomeIcon.C
    )

    private fun populateEditors() {
        val gc = canvas.graphicsContext2D
        editorsInfo.forEach {
            val (_, editor) = it.pair
            editor.init(shapeList, gc, history)
        }
        EmptyEditor.init(shapeList, gc, history)
    }

    fun populate() {
        populateEditors()

        val maps = editorsInfo.associate { it.pair } + (EmptyShape.javaClass to EmptyEdito
        editorHandler = EditorHandler(maps)

        cm.item = CanvasData(shapeList, editorHandler, history, CanvasController(canvas))
        eim.editorsInfo.value = editorsInfo.asObservable()
    }

    fun bindCanvas(pane: Pane) {
        pane += canvas
        canvas.widthProperty().bind(pane.widthProperty())
        canvas.heightProperty().bind(pane.heightProperty())
        canvas.widthProperty().addListener { _, _, _ -> editorHandler.editor.redraw() }
        canvas.heightProperty().addListener { _, _, _ -> editorHandler.editor.redraw() }
    }

    fun undo() = history.undo()
    fun redo() = history.redo()
}
```

Лістинг 15: MainView.kt

```
package com.github.erotourtes.view

import javafx.scene.input.KeyCode
import tornadofx.*

class MainView : View("Lab5") {
    private val ctrl by inject<MainController>()
    override val root = borderpane {
        ctrl.populate()

        top = find<MyMenu>().root
        center = borderpane {
            top = find<ToolBar>().root
            center = pane { ctrl.bindCanvas(this) }
        }

        setOnKeyPressed { event ->
            when {
                event.isControlDown && event.code == KeyCode.Z -> ctrl.undo()
                event.isShiftDown && event.code == KeyCode.Z -> ctrl.redo()
            }
        }
    }
}
```

Лістинг 16: MyMenu.kt

```
package com.github.erotourtes.view

import com.github.erotourtes.drawing.history.NewCommand
import com.github.erotourtes.drawing.history.OpenCommand
import com.github.erotourtes.drawing.shape.ShapeState
import com.github.erotourtes.utils.EditorInfo
import com.github.erotourtes.utils.g
import com.github.erotourtes.utils.n
import com.github.erotourtes.utils.shapeStatesToJSON
import javafx.application.Platform
import javafx.embed.swing.SwingFXUtils
import javafx.scene.control.*
import javafx.stage.FileChooser
import javafx.stage.Modality
import kotlinx.serialization.builtins.ListSerializer
import kotlinx.serialization.json.Json
import tornadofx.*
import java.awt.Desktop
import java.io.File
import javax.imageio.ImageIO
import kotlin.io.path.createTempFile

class MenuController : Controller() {
    private val model by inject<CanvasModel>()
    private val editorsInfoModel by inject<EditorsInfoModel>()

    fun new() {
        with(model) {
            val operation = NewCommand(sl).apply { execute() }
            h.add(operation)
        }
    }

    fun open() {
```

```
    chooseFile()?.let {
        with(model) {
            val shapes = shapeStatesFrom(it).map { it.toShape() }
            OpenCommand(sl, shapes).apply { execute() }.let(h::add)
        }
    }
}

fun saveAs() {
    saveFile()?.writeText(shapeStatesToJSON(model.sl.getStatesList()))
}

fun print() {
    val tempPath = createTempFile(suffix = ".png")
    val image = model.cc.getSnapshotImage()
    ImageIO.write(
        SwingFXUtils.fromFXImage(image, null),
        tempPath.toString().substringAfter("."),
        tempPath.toFile()
    )

    runAsync {
        try {
            val desktop = if (Desktop.isDesktopSupported()) Desktop.getDesktop() else
            desktop?.open(tempPath.toFile())
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}

fun exit() = Platform.exit()

fun create(): Menu {
    val group = ToggleGroup()
    val editorHandler = model.eh
    val list = editorsInfoModel.editorsInfo.value

    editorHandler.listenToChanges { _, _, _ ->
        group.toggles.forEach {
            val userData = it.userData as EditorInfo
            it.isSelected = editorHandler.isCurShapeActive(userData.pair)
        }
    }

    val objectsUI = list.map {
        RadioMenuItem(it.name).apply {
            action { editorHandler.use(it.pair) }
            toggleGroup = group
            isSelected = false
            userData = it
        }
    }
```

```
        return Menu("Objects").apply { items.addAll(objectsUI) }
    }

    fun openTable() {
        find<TableView>().openModal(
            modality = Modality.NONE,
            escapeClosesWindow = true,
            owner = find<MainView>().currentWindow
        )
    }

    private fun saveFile(): File? {
        val fileChooser = FileChooser().apply {
            title = "Save␣as..."
            extensionFilters.addAll(
                FileChooser.ExtensionFilter("JSON", "*.json"),
                FileChooser.ExtensionFilter("ALL", "*.*"),
            )
        }

        return fileChooser.showSaveDialog(find<MainView>().currentWindow)
    }

    private fun chooseFile(): File? {
        val fileChooser = FileChooser().apply {
            title = "Choose␣file..."
            extensionFilters.addAll(
                FileChooser.ExtensionFilter("JSON", "*.json"),
                FileChooser.ExtensionFilter("ALL", "*.*"),
            )
        }

        return fileChooser.showOpenDialog(find<MainView>().currentWindow)
    }

    private fun shapeStatesFrom(it: File): List<ShapeState> {
        val json = Json { prettyPrint = true }
        val jsonShapeState = it.readText()
        return json.decodeFromString(ListSerializer(ShapeState.serializer), jsonShapeState
    }
}

class MyMenu : View() {
    private val ctrl by inject<MenuController>()

    override val root = menubar {
        menu("File") {
            item("New...") { action { ctrl.new() } }
            item("Open...") { action { ctrl.open() } }
            item("Save␣as...") { action { ctrl.saveAs() } }
            separator()
            item("Print") { action { ctrl.print() } }
            separator()
            item("Exit") { action { ctrl.exit() } }
        }

        menus.addAll(ctrl.create())

        menu("Table") { item("Show") { action(ctrl::openTable) } }

        menu("Help") {
            item(
                """
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣0)␣Ж␣=␣$g
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣1)␣Статичнии␣масив␣(Ж␣mod␣3␣!=␣0)␣обсягом␣$g␣+␣100␣=␣$n.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣2)␣"Гумовии"␣слід␣при␣вводі␣об'єктів␣-␣пунктирна␣лінія␣чорного␣кольору␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣3)␣Прямокутник:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Увід␣прямокутника:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣від␣центру␣до␣одного␣з␣кутів␣для␣(Ж␣mod␣2␣=␣1)␣g␣%␣2␣=␣${g␣%␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Відображення␣прямокутника:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣чорнии␣контур␣прямокутника␣без␣заповнення␣для␣(Ж␣mod␣5␣=␣3␣а
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Кольори␣заповнення␣прямокутника:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣сірии␣для␣(Ж␣mod␣6␣=␣5)␣g␣%␣6␣=␣${g␣%␣6}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣4)␣Еліпс:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Ввід␣еліпсу:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣по␣двом␣протилежним␣кутам␣охоплюючого␣прямокутника␣для␣варіа
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Відображення␣еліпсу:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣чорнии␣контур␣з␣кольоровим␣заповненням␣для␣(Ж␣mod␣5␣=␣3␣або␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Кольори␣заповнення␣еліпсу:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣померанчевии␣для␣(Ж␣mod␣6␣=␣5)␣g␣%␣6␣=␣${g␣%␣6}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣5)␣Позначка␣поточного␣типу␣об'єкту,␣що␣вводиться
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣в␣заголовку␣вікна␣для␣(Ж␣mod␣2␣=␣1)␣g␣%␣2␣=␣${g␣%␣2}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣""".trimIndent()
            )
        }
    }
}
```

Лістинг 17: MyModel.kt

```
package com.github.erotourtes.view

import com.github.erotourtes.drawing.CanvasController
import com.github.erotourtes.drawing.EditorHandler
import com.github.erotourtes.drawing.history.History
import com.github.erotourtes.drawing.editor.ShapesList
import com.github.erotourtes.utils.EditorInfo
import javafx.beans.property.SimpleListProperty
import tornadofx.*

data class CanvasData(
    val shapesList: ShapesList,
    val editorHandler: EditorHandler,
    val history: History,
    val canvasController: CanvasController,
)

class CanvasModel : ItemViewModel<CanvasData>() {
    private val shapesList = bind(CanvasData::shapesList)
    private val editorHandler = bind(CanvasData::editorHandler)
    private val history = bind(CanvasData::history)
    private val canvasController = bind(CanvasData::canvasController)

    val eh: EditorHandler by editorHandler
    val sl: ShapesList by shapesList
    val h: History by history
    val cc: CanvasController by canvasController
}

class EditorsInfoData {
    val editorsInfo = SimpleListProperty<EditorInfo>()
}

class EditorsInfoModel : ItemViewModel<EditorsInfoData>() {
    val editorsInfo = bind(EditorsInfoData::editorsInfo)
}
```

Лістинг 18: TableView.kt

```
package com.github.erotourtes.view

import com.github.erotourtes.drawing.history.ChangeCoordinatesCommand
import com.github.erotourtes.drawing.history.RemoveItemCommand
import com.github.erotourtes.drawing.shape.EmptyShape
import com.github.erotourtes.drawing.shape.Shape
import com.github.erotourtes.drawing.shape.Shape.ShapeModel
import com.github.erotourtes.utils.shapeStatesToJSON
import javafx.beans.property.SimpleStringProperty
import javafx.beans.value.ObservableValue
import javafx.collections.ListChangeListener
import javafx.collections.ObservableList
import javafx.geometry.Pos
import javafx.scene.control.TableView
import javafx.stage.FileChooser
import javafx.stage.StageStyle
import tornadofx.*
import java.io.File

class Table<E, S>(
    private val data: ObservableList<E>,
    private val columnsData: List<Pair<String, (E) -> ObservableValue<S>>>,
) {

    var onUserSelectCb: (E) -> Unit = {}

    val root
        get() = TableView<E>().apply {
            columnsData.forEach {
                val (name, gerObservable) = it
                column(name) { value -> gerObservable(value.value) }
            }

            items = data
            onUserSelect(1, onUserSelectCb)
        }
}

class Form : View("Edit") {
    private val model by inject<CanvasModel>()
    private val shapeModel by inject<ShapeModel>()

    override val root = form {
        val editorHandler = model.eh
        hiddenWhen(shapeModel.isEmptyShape)
        fieldset("Selected␣Shape") {
            textProperty.bind(shapeModel.itemProperty.stringBinding { "Selected␣Shape␣${it

            hbox {
                style { alignment = Pos.CENTER }
                button("Save") {
                    enableWhen(shapeModel.dirty)
                    action {
                        // TODO: bind redraw to changes in coordinates
                        ChangeCoordinatesCommand(shapeModel, editorHandler::requestRedraw)
                            execute()
                            model.h.add(this)
                        }
                    }
                }
                button("Reset") { action { shapeModel.rollback() } }
                button("Delete") {
                    action {
                        RemoveItemCommand(model.sl, shapeModel.item).apply {
                            execute()
                            model.h.add(this)
                        }
                        this@Form.close()
                    }
                }
                button("Close") {
                    action {
                        shapeModel.item = EmptyShape
                        this@Form.close()
                    }
                }
            }
            field("x1:␣") { textfield(shapeModel.x1) }
            field("x2:␣") { textfield(shapeModel.x2) }
            field("y1:␣") { textfield(shapeModel.y1) }
            field("y2:␣") { textfield(shapeModel.y2) }
        }
    }
}

class TableController : Controller() {
    private val model by inject<CanvasModel>()
    private var file: File? = null
    private val listener = ListChangeListener<Shape> {
        file?.writeText(shapeStatesToJSON(model.sl.getStatesList()))
    }

    var fileNameProp = SimpleStringProperty()
    private var fileName by fileNameProp

    val data get() = model.sl.getObservableList()

    fun highlight(shape: Shape) {
        model.eh.editor.highlight(shape)
    }

    fun selectFile() {
        file = getFile()
        fileName = file?.name ?: "No␣file␣selected"
    }

    fun autoSave(isSelected: Boolean) {
        if (isSelected) {
            data.addListener(listener)
            listener.onChanged(null)
        } else data.removeListener(listener)
    }

    private fun getFile() = chooseFile(
        "Choose␣file...",
        filters = arrayOf(
            FileChooser.ExtensionFilter("JSON", "*.json"),
            FileChooser.ExtensionFilter("ALL", "*.*"),
        ),
        mode = FileChooserMode.Single,
    ).firstOrNull()
}

class TableView : View("Table") {
    private val ctrl by inject<TableController>()
    private val shapeModel by inject<ShapeModel>()

    private val columnsData = listOf(
        "x1" to { shape: Shape -> shape.x1Prop },
        "y1" to { shape: Shape -> shape.y1Prop },
        "x2" to { shape: Shape -> shape.x2Prop },
        "y2" to { shape: Shape -> shape.y2Prop },
    )

    private val table = Table(ctrl.data, columnsData).apply {
        onUserSelectCb = {
            find<Form>().openModal(
                stageStyle = StageStyle.UTILITY,
                escapeClosesWindow = true,
                owner = this@TableView.currentWindow
            )

            ctrl.highlight(it)
            shapeModel.item = it
        }
    }

    override val root = borderpane {
        center = table.root
        bottom = hbox {
            button("Select␣File") { action { ctrl.selectFile() } }
            checkbox("Auto␣Save␣to␣file") {
                bind(ctrl.fileNameProp.stringBinding { str -> "Auto␣save␣to␣file␣${str␣?:␣'
                action { ctrl.autoSave(isSelected) }
            }
        }
    }
}
```

Лістинг 19: ToolBar.kt

```
package com.github.erotourtes.view

import com.github.erotourtes.drawing.editor.EmptyEditor
import com.github.erotourtes.styles.ToolbarStyles
import com.github.erotourtes.utils.EditorInfo
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIcon
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIconView
import javafx.scene.control.SpinnerValueFactory
import javafx.scene.control.ToggleButton
import javafx.scene.control.ToggleGroup
import javafx.scene.layout.BorderPane
import javafx.scene.paint.Color
import javafx.stage.StageStyle
import tornadofx.*

class ToolBarController : Controller() {
    val detached
        get() = isDetached
    private val model by inject<CanvasModel>()
    private val editorsInfoModel by inject<EditorsInfoModel>()
    private val group = ToggleGroup()
    private lateinit var toolBar: ToolBar
    private var isDetached = false

    fun setView(view: ToolBar) {
        this.toolBar = view
    }

    fun create() = editorsInfoModel.editorsInfo.value.map {
        ToggleButton().apply {
            tooltip(it.tooltip)
            addClass(ToolbarStyles.iconButton)
            add(FontAwesomeIconView(it.icon).apply { addClass(ToolbarStyles.icon) })
            toggleGroup = group
            isSelected = false
            userData = it
            action {
                val eh = model.eh

                if (this.isSelected) eh.use(it.pair)
                else eh.use(Pair(EmptyEditor.shape.javaClass, EmptyEditor))
            }
        }
    }

    fun listenToEditorChange() {
        val editorHandler = model.eh

        editorHandler.listenToChanges { _, _, _ ->
            group.toggles.forEach {
                val userData = it.userData as EditorInfo
                it.isSelected = editorHandler.isCurShapeActive(userData.pair)
            }
        }
    }

    fun undo() = model.h.undo()
    fun redo() = model.h.redo()

    fun toggle() = if (isDetached) attach() else detach()

    fun changeMainColor(color: Color) = model.cc.changeFillColor(color)

    val mainColorProp = model.cc.fillColorProp
    val auxColorProp = model.cc.strokeColorProp

    fun changeAuxColor(color: Color) = model.cc.changeStrokeColor(color)

    fun changeStroke(width: Double) = model.cc.changeStrokeWidth(width)

    fun swapColors() {
        val aux = auxColorProp.value
        val main = mainColorProp.value
        model.cc.also {
            it.changeStrokeColor(main)
            it.changeFillColor(aux)
        }
    }

    private fun attach() {
        isDetached = false
        val center = find<MainView>().root.center as BorderPane
        toolBar.close()
        center.top = toolBar.root
    }

    private fun detach() {
        isDetached = true
        toolBar.removeFromParent()
        toolBar.openWindow(StageStyle.UTILITY)
    }
```

```
}

class ToolBar : View() {
    private val ctrl by inject<ToolBarController>()

    override fun onDock() {
        ctrl.setView(this)
        ctrl.listenToEditorChange()
    }

    override val root = flowpane {
        addClass(ToolbarStyles.toolbar)

        button {
            addClass(ToolbarStyles.iconButton)
            val detach = FontAwesomeIconView(FontAwesomeIcon.UNLINK)
            val attach = FontAwesomeIconView(FontAwesomeIcon.LINK)
            graphic = detach

            action {
                graphic = if (ctrl.detached) detach else attach
                ctrl.toggle()
            }
        }
        button {
            addClass(ToolbarStyles.iconButton)
            graphic = FontAwesomeIconView(FontAwesomeIcon.REPEAT)

            action { ctrl.redo() }
        }
        button {
            addClass(ToolbarStyles.iconButton)
            graphic = FontAwesomeIconView(FontAwesomeIcon.UNDO)

            action { ctrl.undo() }
        }

        label {
            style { borderWidth += box(0.px, 0.px, 0.px, 1.px); borderColor += box(c("#000(
        }

        ctrl.create().forEach { add(it) }

        colorpicker {
            addClass(ToolbarStyles.iconButton)
            valueProperty().bindBidirectional(ctrl.mainColorProp)
            setOnAction { ctrl.changeMainColor(value) }
        }

        colorpicker {
            addClass(ToolbarStyles.iconButton)
            valueProperty().bindBidirectional(ctrl.auxColorProp)
            setOnAction { ctrl.changeAuxColor(value) }
        }
        button {
            addClass(ToolbarStyles.iconButton)
            graphic = FontAwesomeIconView(FontAwesomeIcon.EXCHANGE)

            action { ctrl.swapColors() }
        }

        spinner<Double> {
            addClass(ToolbarStyles.iconButton)
            valueFactory = SpinnerValueFactory.DoubleSpinnerValueFactory(1.0, 10.0, 1.0)
            valueProperty().addListener { _, _, new -> ctrl.changeStroke(new) }
        }
    }
}
```

3.6 Module: com.github.erotourtes.utils

Лістинг 20: Dimension.kt

```
package com.github.erotourtes.utils

import javafx.beans.property.SimpleDoubleProperty
import kotlinx.serialization.Serializable
import tornadofx.*
import kotlin.math.abs

@Serializable
class Dimension {
    private val x1Prop = SimpleDoubleProperty(0.0)
    private val y1Prop = SimpleDoubleProperty(0.0)
    private val x2Prop = SimpleDoubleProperty(0.0)
    private val y2Prop = SimpleDoubleProperty(0.0)

    private var x1 by x1Prop
    private var y1 by y1Prop
    private var x2 by x2Prop
    private var y2 by y2Prop

    val getX1Prop get() = x1Prop
    val getY1Prop get() = y1Prop
    val getX2Prop get() = x2Prop
    val getY2Prop get() = y2Prop

    val width: Double get() = abs(x2 - x1)
    val height: Double get() = abs(y2 - y1)

    fun setStart(x: Double, y: Double): Dimension {
        x1 = x
        y1 = y
        return this
    }

    fun setEnd(x: Double, y: Double): Dimension {
        x2 = x
        y2 = y
        return this
    }

    fun copyTo(dst: Dimension) {
        dst.x1 = x1
        dst.y1 = y1
        dst.x2 = x2
        dst.y2 = y2
    }

    fun copyFrom(src: Dimension) = src.copyTo(this)

    fun copy(): Dimension = Dimension().apply { copyFrom(this@Dimension) }

    fun getBoundaries(): Pair<Point, Point> {
        return Pair(
            Point(x1.coerceAtMost(x2), y1.coerceAtMost(y2)), Point(x1.coerceAtLeast(x2), y1
        )
    }

    fun getRaw(): Pair<Point, Point> {
        return Pair(
            Point(x1, y1), Point(x2, y2)
        )
    }

    data class Point(val x: Double, val y: Double)

    override fun toString(): String = "Dimension(x1=$x1, y1=$y1, x2=$x2, y2=$y2)"

    companion object {
        fun toCorner(dm: Dimension): Dimension {
            val (c, end) = dm.getRaw()

            // it is not width, it is half of the width; can be negative
            val w = end.x - c.x
            val h = end.y - c.y

            val sX = c.x - w
            val sY = c.y - h

            return Dimension().setStart(end.x, end.y).setEnd(sX, sY)
        }

        fun toEqual(dm: Dimension): Dimension {
            val (s, e) = dm.getRaw()
            val w = e.x - s.x
            val h = e.y - s.y
            val size = abs(w).coerceAtLeast(abs(h))
            val safeW = if (w == 0.0) 1.0 else w
            val safeH = if (h == 0.0) 1.0 else h
            val normalizedX = safeW / abs(safeW) * size
            val normalizedY = safeH / abs(safeH) * size
            return Dimension().setStart(s.x, s.y).setEnd(s.x + normalizedX, s.y + normalize
        }

        fun from(x1: Double, y1: Double, x2: Double, y2: Double): Dimension =
            Dimension().setStart(x1, y1).setEnd(x2, y2)

        val serializer get() = DimensionSerializer
    }
}
```

Лістинг 21: ExtensionFunctions.kt

```
package com.github.erotourtes.utils

import javafx.scene.canvas.GraphicsContext


fun GraphicsContext.fillRect(dm: Dimension) {
    val (s, e) = dm.getBoundaries()
    fillRect(s.x, s.y, e.x - s.x, e.y - s.y)
}

fun GraphicsContext.strokeRect(dm: Dimension) {
    val (s, e) = dm.getBoundaries()
    strokeRect(s.x, s.y, e.x - s.x, e.y - s.y)
}

fun GraphicsContext.fillOval(dm: Dimension) {
    val (s, e) = dm.getBoundaries()
    fillOval(s.x, s.y, e.x - s.x, e.y - s.y)
}

fun GraphicsContext.strokeOval(dm: Dimension) {
    val (s, e) = dm.getBoundaries()
    strokeOval(s.x, s.y, e.x - s.x, e.y - s.y)
}

fun GraphicsContext.strokeLine(dm: Dimension) {
    val (s, e) = dm.getRaw()
    strokeLine(s.x, s.y, e.x, e.y)
}

inline fun GraphicsContext.drawOnce(lambda: GraphicsContext.() -> Unit) {
    save()
    lambda(this)
    restore()
}
```

Лістинг 22: Serializers.kt

```
package com.github.erotourtes.utils

import com.github.erotourtes.drawing.GCState
import com.github.erotourtes.drawing.shape.ShapeState
```

```kotlin
import javafx.scene.paint.Color
import kotlinx.serialization.KSerializer
import kotlinx.serialization.Serializer
import kotlinx.serialization.builtins.ListSerializer
import kotlinx.serialization.builtins.serializer
import kotlinx.serialization.descriptors.SerialDescriptor
import kotlinx.serialization.descriptors.buildClassSerialDescriptor
import kotlinx.serialization.encoding.CompositeDecoder
import kotlinx.serialization.encoding.Decoder
import kotlinx.serialization.encoding.Encoder

@Serializer(forClass = ShapeState::class)
object ShapeStateSerializer : KSerializer<ShapeState> {
    override val descriptor: SerialDescriptor = buildClassSerialDescriptor("ShapeState") {
        element("className", String.serializer().descriptor)
        element("dm", Dimension.serializer.descriptor)
        element("gcState", GCState.serializer.descriptor)
    }

    override fun deserialize(decoder: Decoder): ShapeState {
        val composite = decoder.beginStructure(descriptor)
        var className: String? = null
        var dm: Dimension? = null
        var gcState: GCState? = null

        loop@ while (true) {
            when (val i = composite.decodeElementIndex(descriptor)) {
                CompositeDecoder.DECODE_DONE -> break@loop
                0 -> className = composite.decodeStringElement(descriptor, i)
                1 -> dm = composite.decodeSerializableElement(descriptor, i, Dimension.ser
                2 -> gcState =
                    composite.decodeSerializableElement(descriptor, i, GCState.serializer)

                else -> throw IllegalArgumentException("Unexpected␣index:␣$i")
            }
        }

        composite.endStructure(descriptor)

        return ShapeState(
            className!!,
            dm!!,
            gcState!!
        )
    }

    override fun serialize(encoder: Encoder, value: ShapeState) {
        val composite = encoder.beginStructure(descriptor)
        composite.encodeStringElement(descriptor, 0, value.clazz)
        composite.encodeSerializableElement(descriptor, 1, Dimension.serializer, value.dm)
        composite.encodeSerializableElement(descriptor, 2, GCState.serializer, value.gcStat
        composite.endStructure(descriptor)
    }
}
```

```
@Serializer(forClass = ShapeState::class)
object GCStateSerializer : KSerializer<GCState> {
    override val descriptor: SerialDescriptor = buildClassSerialDescriptor("GCState") {
        element("fill", String.serializer().descriptor)
        element("stroke", String.serializer().descriptor)
        element("lineWidth", Double.serializer().descriptor)
    }

    override fun deserialize(decoder: Decoder): GCState {
        val composite = decoder.beginStructure(descriptor)

        var fill: String? = null
        var stroke: String? = null
        var lineWidth: Double? = null

        loop@ while (true) {
            when (val i = composite.decodeElementIndex(descriptor)) {
                CompositeDecoder.DECODE_DONE -> break@loop
                0 -> fill = composite.decodeStringElement(descriptor, i)
                1 -> stroke = composite.decodeStringElement(descriptor, i)
                2 -> lineWidth = composite.decodeDoubleElement(descriptor, i)
                else -> throw IllegalArgumentException("Unexpected␣index:␣$i")
            }
        }

        composite.endStructure(descriptor)

        return GCState(
            Color.valueOf(fill!!),
            Color.valueOf(stroke!!),
            lineWidth!!,
        )
    }

    override fun serialize(encoder: Encoder, value: GCState) {
        val composite = encoder.beginStructure(descriptor)
        composite.encodeStringElement(descriptor, 0, value.fill.toString())
        composite.encodeStringElement(descriptor, 1, value.stroke.toString())
        composite.encodeDoubleElement(descriptor, 2, value.lineWidth)
        composite.endStructure(descriptor)
    }
}

@Serializer(forClass = Dimension::class)
object DimensionSerializer : KSerializer<Dimension> {
    override val descriptor: SerialDescriptor = ListSerializer(Double.serializer()).descrip

    override fun deserialize(decoder: Decoder): Dimension {
        val list = decoder.decodeSerializableValue(ListSerializer(Double.serializer()))

        return Dimension().apply {
            val (x1, y1, x2, y2) = list
            setStart(x1, y1)
            setEnd(x2, y2)
        }
    }

    override fun serialize(encoder: Encoder, value: Dimension) {
        val list = value.getRaw().toList().map { (x, y) -> listOf(x, y) }.flatten()
        encoder.encodeSerializableValue(ListSerializer(Double.serializer()), list)
    }
}
```

Лістинг 23: Utils.kt

```
package com.github.erotourtes.utils

import com.github.erotourtes.drawing.editor.DmProcessor
import com.github.erotourtes.drawing.editor.Editor
import com.github.erotourtes.drawing.shape.Shape
import com.github.erotourtes.drawing.shape.ShapeState
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIcon
import kotlinx.serialization.builtins.ListSerializer
import kotlinx.serialization.json.Json

data class EditorInfo(
    val name: String,
    val tooltip: String,
    val pair: Pair<Class<out Shape>, Editor>,
    // [icons](https://fontawesome.com/v4/icons/)
    var icon: FontAwesomeIcon? = null,
)

const val g = 22 + 1
const val n = 100 + g

fun pipe(vararg processors: DmProcessor): DmProcessor = DmProcessor { dm ->
    processors.fold(dm) { acc, processor -> processor.process(acc) }
}

open class SingletonHolder<out T>(private val constructor: () -> T) {

    @Volatile
    private var instance: T? = null

    fun getInstance(): T =
        instance ?: synchronized(this) {
            instance ?: constructor().also { instance = it }
        }
}

fun shapeStatesToJSON(list: List<ShapeState>): String {
    val json = Json { prettyPrint = true }
    return json.encodeToString(ListSerializer(ShapeState.serializer), list)
}
```

3.7 Module: com.github.erotourtes.styles

Лістинг 24: ToolbarStyles.kt

```
package com.github.erotourtes.styles

import tornadofx.*
import javafx.scene.paint.Color

class ToolbarStyles : Stylesheet() {

    companion object {
        val toolbar by cssclass()
        val iconButton by cssclass()
        val icon by cssclass()
        val dark = c("#555")
        val light = Color.LIGHTSTEELBLUE!!
    }

    init {
        val toolbarHeight = 40.px
        toolbar {
            padding = box(5.px)
            spacing = 5.px
            minHeight = toolbarHeight
            alignment = javafx.geometry.Pos.CENTER_LEFT
            borderWidth += box(0.px, 0.px, 1.px, 0.px)
            borderColor += box(dark)
        }

        iconButton {
            backgroundColor += Color.TRANSPARENT

            and(selected) {
                backgroundColor += dark
                icon { fill = light }
            }

            minHeight = toolbarHeight / 1.5
            maxHeight = toolbarHeight / 1.5
        }

        icon { fill = dark }
    }
}
```

3.8 Module: com.github.erotourtes.app

Лістинг 25: MyApp.kt

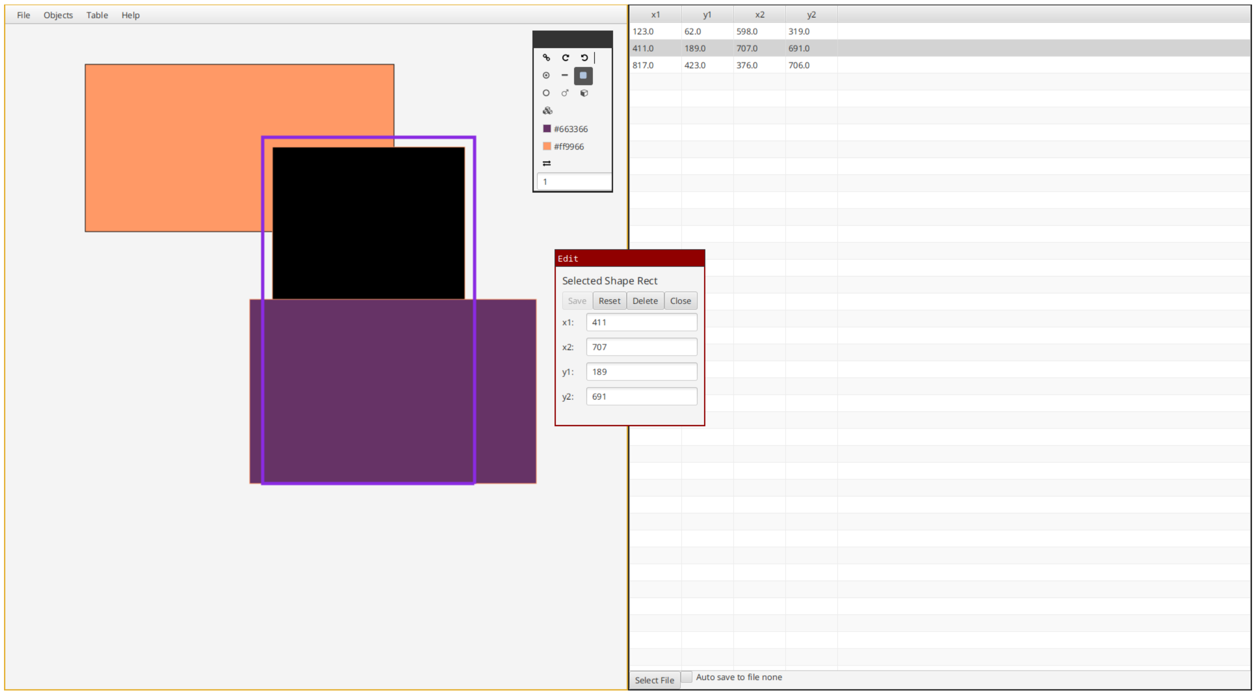
```
package com.github.erotourtes.app

import com.github.erotourtes.styles.ToolbarStyles
import com.github.erotourtes.view.MainView
import tornadofx.App

class MyApp: App(MainView::class, ToolbarStyles::class)
```

4 Ілюстрації:

4.1 Images



x1	y1	x2	y2
123.0	62.0	598.0	319.0
411.0	189.0	707.0	691.0
817.0	423.0	376.0	706.0

Edit

Selected Shape Rect

Save Reset Delete Close

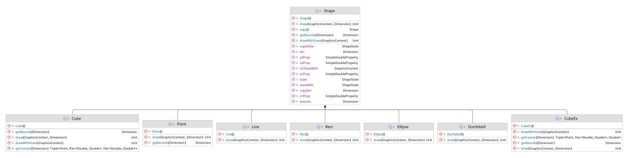
x1: 411

x2: 707

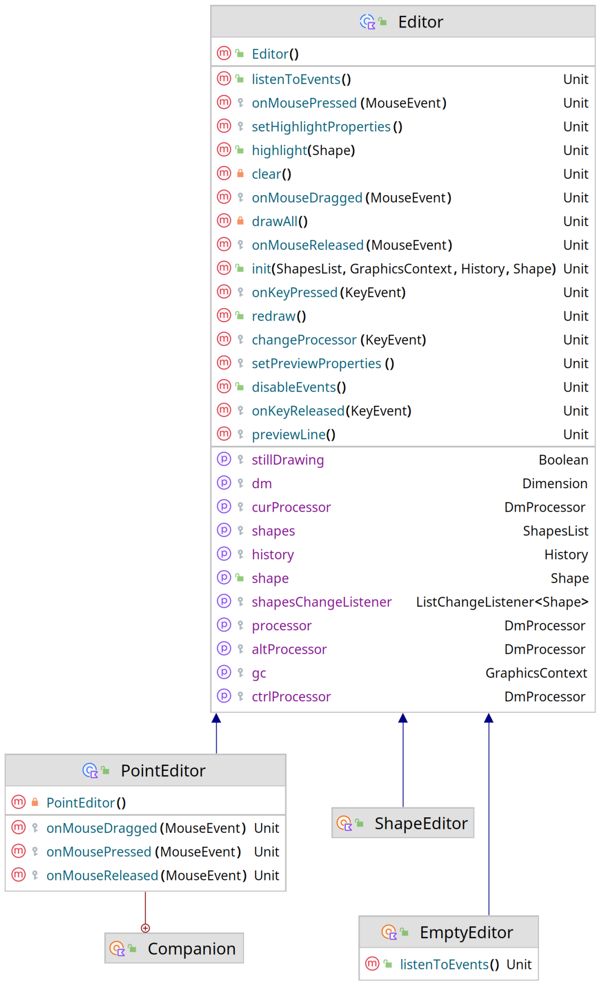
y1: 189

y2: 691

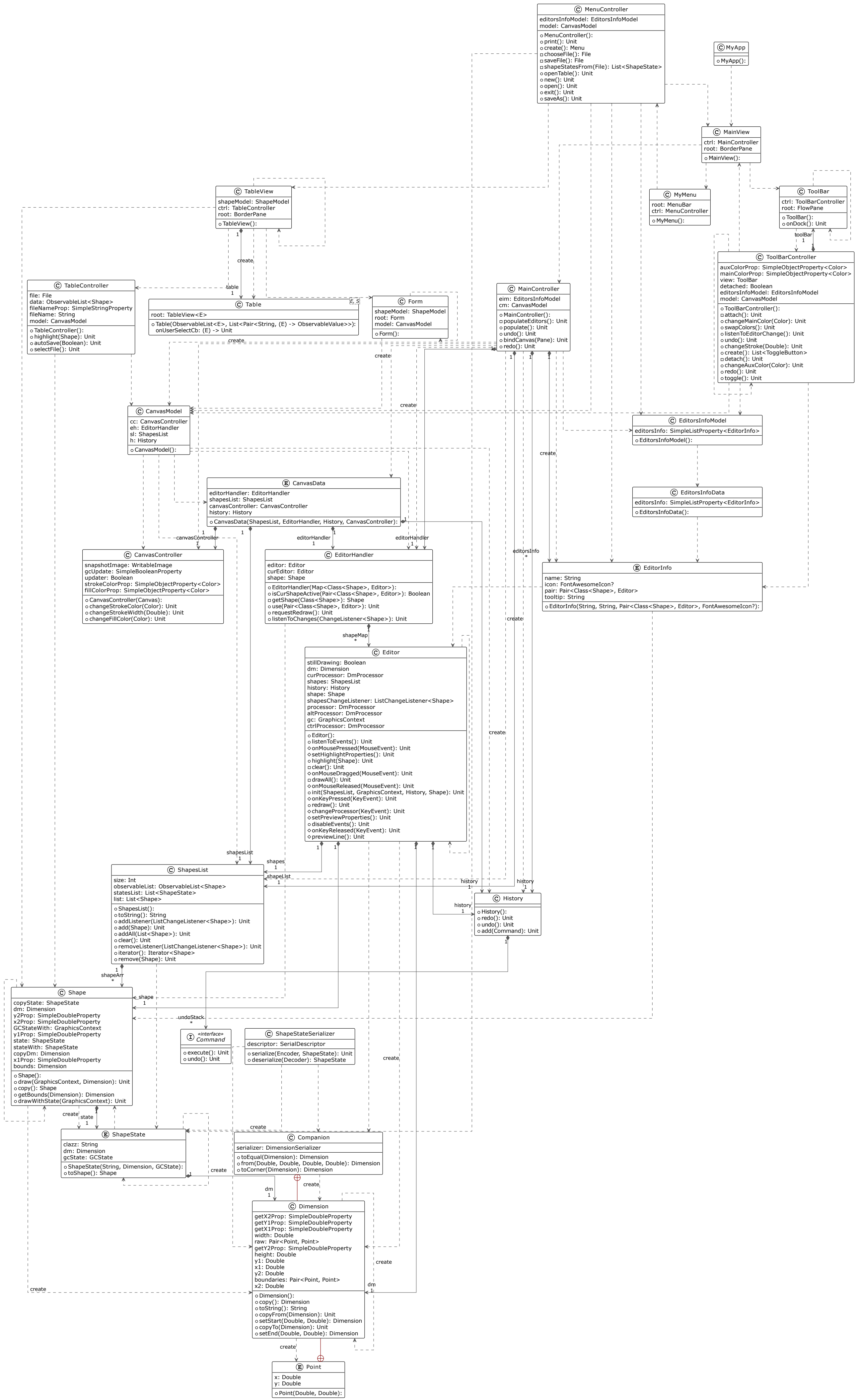
4.2 UML


Shape
Shape()
draw(GraphicsContext, Dimension) Unit
copy() Shape
getBounds(Dimension) Dimension
drawWithState(GraphicsContext) Unit
copyState ShapeState
dm Dimension
y2Prop SimpleDoubleProperty
x2Prop SimpleDoubleProperty
GCStateWith GraphicsContext
y1Prop SimpleDoubleProperty
state ShapeState
stateWith ShapeState
copyDm Dimension
x1Prop SimpleDoubleProperty
bounds Dimension
Cube
Cube()
getBounds(Dimension) Dimension
draw(GraphicsContext, Dimension) Unit
drawWithState(GraphicsContext) Unit
getCoords(Dimension) Triple<Point, Pair<Double, Double>, Pair<Double, Double>>
Point
Point()
draw(GraphicsContext, Dimension) Unit
getBounds(Dimension) Dimension
Line
Line()
draw(GraphicsContext, Dimension) Unit
Rect
Rect()
draw(GraphicsContext, Dimension) Unit
Ellipse
Ellipse()
draw(GraphicsContext, Dimension) Unit
Dumbbell
Dumbbell()
draw(GraphicsContext, Dimension) Unit
CubeEx
CubeEx()
drawWithState(GraphicsContext) Unit
getCoords(Dimension) Triple<Point, Pair<Double, Double>, Double>
getBounds(Dimension) Dimension
draw(GraphicsContext, Dimension) Unit

(a) Shapes


Editor
Editor()
listenToEvents() Unit
onMousePressed(MouseEvent) Unit
setHighlightProperties() Unit
highlight(Shape) Unit
clear() Unit
onMouseDragged(MouseEvent) Unit
drawAll() Unit
onMouseReleased(MouseEvent) Unit
init(ShapesList, GraphicsContext, History, Shape) Unit
onKeyPressed(KeyEvent) Unit
redraw() Unit
changeProcessor(KeyEvent) Unit
setPreviewProperties() Unit
disableEvents() Unit
onKeyReleased(KeyEvent) Unit
previewLine() Unit
stillDrawing Boolean
dm Dimension
curProcessor DmProcessor
shapes ShapesList
history History
shape Shape
shapesChangeListener ListChangeListener<Shape>
processor DmProcessor
altProcessor DmProcessor
gc GraphicsContext
ctrlProcessor DmProcessor
PointEditor
PointEditor()
onMouseDragged(MouseEvent) Unit
onMousePressed(MouseEvent) Unit
onMouseReleased(MouseEvent) Unit
Companion
ShapeEditor
EmptyEditor
listenToEvents() Unit

(б) Editors


MenuController
editorsInfoModel: EditorsInfoModel
model: CanvasModel
MenuController():
print(): Unit
create(): Menu
chooseFile(): File
saveFile(): File
shapeStatesFrom(File): List<ShapeState>
openTable(): Unit
new(): Unit
open(): Unit
exit(): Unit
saveAs(): Unit
MyApp
MyApp():
MainView
ctrl: MainController
root: BorderPane
MainView():
TableView
shapeModel: ShapeModel
ctrl: TableController
root: BorderPane
TableView():
MyMenu
root: MenuBar
ctrl: MenuController
MyMenu():
ToolBar
ctrl: ToolBarController
root: FlowPane
ToolBar():
onDock(): Unit
toolBar
ToolBarController
auxColorProp: SimpleObjectProperty<Color>
mainColorProp: SimpleObjectProperty<Color>
view: ToolBar
detached: Boolean
editorsInfoModel: EditorsInfoModel
model: CanvasModel
ToolBarController():
attach(): Unit
changeMainColor(Color): Unit
swapColors(): Unit
listenToEditorChange(): Unit
undo(): Unit
changeStroke(Double): Unit
create(): List<ToggleButton>
detach(): Unit
changeAuxColor(Color): Unit
redo(): Unit
toggle(): Unit
TableController
file: File
data: ObservableList<Shape>
fileNameProp: SimpleStringProperty
fileName: String
model: CanvasModel
TableController():
highlight(Shape): Unit
autoSave(Boolean): Unit
selectFile(): Unit
table
create
Table
E, S
root: TableView<E>
Table(ObservableList<E>, List<Pair<String, (E) -> ObservableValue>>):
onUserSelectCb: (E) -> Unit
Form
shapeModel: ShapeModel
root: Form
model: CanvasModel
Form():
MainController
eim: EditorsInfoModel
cm: CanvasModel
MainController():
populateEditors(): Unit
populate(): Unit
undo(): Unit
bindCanvas(Pane): Unit
redo(): Unit
CanvasModel
cc: CanvasController
eh: EditorHandler
sl: ShapesList
h: History
CanvasModel():
EditorsInfoModel
editorsInfo: SimpleListProperty<EditorInfo>
EditorsInfoModel():
CanvasData
editorHandler: EditorHandler
shapesList: ShapesList
canvasController: CanvasController
history: History
CanvasData(ShapesList, EditorHandler, History, CanvasController):
EditorsInfoData
editorsInfo: SimpleListProperty<EditorInfo>
EditorsInfoData():
canvasController
editorHandler
editorsInfo
CanvasController
snapshotImage: WritableImage
gcUpdate: SimpleBooleanProperty
updater: Boolean
strokeColorProp: SimpleObjectProperty<Color>
fillColorProp: SimpleObjectProperty<Color>
CanvasController(Canvas):
changeStrokeColor(Color): Unit
changeStrokeWidth(Double): Unit
changeFillColor(Color): Unit
EditorHandler
editor: Editor
curEditor: Editor
shape: Shape
EditorHandler(Map<Class<Shape>, Editor>):
isCurShapeActive(Pair<Class<Shape>, Editor>): Boolean
getShape(Class<Shape>): Shape
use(Pair<Class<Shape>, Editor>): Unit
requestRedraw(): Unit
listenToChanges(ChangeListener<Shape>): Unit
EditorInfo
name: String
icon: FontAwesomeIcon?
pair: Pair<Class<Shape>, Editor>
tooltip: String
EditorInfo(String, String, Pair<Class<Shape>, Editor>, FontAwesomeIcon?):
shapeMap
Editor
stillDrawing: Boolean
dm: Dimension
curProcessor: DmProcessor
shapes: ShapesList
history: History
shape: Shape
shapesChangeListener: ListChangeListener<Shape>
processor: DmProcessor
altProcessor: DmProcessor
gc: GraphicsContext
ctrlProcessor: DmProcessor
Editor():
listenToEvents(): Unit
onMousePressed(MouseEvent): Unit
setHighlightProperties(): Unit
highlight(Shape): Unit
clear(): Unit
onMouseDragged(MouseEvent): Unit
drawAll(): Unit
onMouseReleased(MouseEvent): Unit
init(ShapesList, GraphicsContext, History, Shape): Unit
onKeyPressed(KeyEvent): Unit
redraw(): Unit
changeProcessor(KeyEvent): Unit
setPreviewProperties(): Unit
disableEvents(): Unit
onKeyReleased(KeyEvent): Unit
previewLine(): Unit
shapesList
shapes
shapeList
history
ShapesList
size: Int
observableList: ObservableList<Shape>
statesList: List<ShapeState>
list: List<Shape>
ShapesList():
toString(): String
addListener(ListChangeListener<Shape>): Unit
add(Shape): Unit
addAll(List<Shape>): Unit
clear(): Unit
removeListener(ListChangeListener<Shape>): Unit
iterator(): Iterator<Shape>
remove(Shape): Unit
History
History():
redo(): Unit
undo(): Unit
add(Command): Unit
shapeArr
shape
undoStack
Shape
copyState: ShapeState
dm: Dimension
y2Prop: SimpleDoubleProperty
x2Prop: SimpleDoubleProperty
GCStateWith: GraphicsContext
y1Prop: SimpleDoubleProperty
state: ShapeState
stateWith: ShapeState
copyDm: Dimension
x1Prop: SimpleDoubleProperty
bounds: Dimension
Shape():
draw(GraphicsContext, Dimension): Unit
copy(): Shape
getBounds(Dimension): Dimension
drawWithState(GraphicsContext): Unit
«interface»
Command
execute(): Unit
undo(): Unit
ShapeStateSerializer
descriptor: SerialDescriptor
serialize(Encoder, ShapeState): Unit
deserialize(Decoder): ShapeState
state
ShapeState
clazz: String
dm: Dimension
gcState: GCState
ShapeState(String, Dimension, GCState):
toShape(): Shape
Companion
serializer: DimensionSerializer
toEqual(Dimension): Dimension
from(Double, Double, Double, Double): Dimension
toCorner(Dimension): Dimension
dm
Dimension
getX2Prop: SimpleDoubleProperty
getY1Prop: SimpleDoubleProperty
getX1Prop: SimpleDoubleProperty
width: Double
raw: Pair<Point, Point>
getY2Prop: SimpleDoubleProperty
height: Double
y1: Double
x1: Double
y2: Double
boundaries: Pair<Point, Point>
x2: Double
Dimension():
copy(): Dimension
toString(): String
copyFrom(Dimension): Unit
setStart(Double, Double): Dimension
copyTo(Dimension): Unit
setEnd(Double, Double): Dimension
Point
x: Double
y: Double
Point(Double, Double):

5 Висновки:

Отже, я отримав вміння та навички проектування багатовіконних програм на C++ в об'єктно-орієнтованому стилі.