1　概要

Java の awt 及び swing を用いてドローエディタを作成した.
効率的に作業できるドローエディタを目指して作成した. 具体的には以下の 2 点を意識した. [*1]

- マウスはキャンバス上から極力動かさない.
- 左手はキーボードのホームポジションから極力動かさない.

プログラムの設計として, クラスが持つ役割をできるだけ分割して, 明確にすることを目指した. きっちりと役割分担することで, 機能を追加したいときに, どこを変更すればよいか分かりやすくした.

1.1　実装した機能一覧

以下に実装した機能を一覧で示す, 具体的な内容は付録 A や 4 章の実行例を参照.

- コマンド操作機能
  - マウス: キャンバス上の座標指定とダイアログの操作
  - キー: 座標指定以外の指示を行う
- コマンドナビゲータの表示
- 3 色保持できるパレット
- 色変更機能 (JColorChooser)
- 5 種のブラシ
  - 枠線だけの四角
  - 塗りつぶされた四角
  - 枠線だけの楕円
  - 塗りつぶされた楕円
  - 直線
- 選択機能
  - 選択した図形の削除
- Undo・Redo のような機能
  - Undo: 選択された図形をゴミ箱スタックに移動させる
  - Redo: ゴミ箱スタックから描画スタックに移動させる

1.2　主な未実装の機能

- 各種保存機能
- 一般的な Undo・Redo 機能
- キャンバスサイズ固定機能
- マクロ機能

[*1] 右手でマウス, 左手でキーボードを操作する想定で作成した.

2　設計方針

機能を追加するときにどのクラスを変更すればよいかを明確することを目指して設計した．そのため, 各クラスの役割を明確にし, できるだけコードを分割した．

以下では, まず, 全体の設計について述べる．その後, このプログラムで最も重要なコマンドシステムの設計について述べる．最後に, ドローエディタの主機能である図形の描画についての設計について述べる．

デザインパターンとしては, MVC(Model, View, Controller) モデルをベースにした．ソフトウェアのコアとなる部分, ユーザーに見える部分, ユーザーが触れる部分に分離することで, 少ない変更でソフトウェアの見た目だけ変えたり, 操作方法を変えたりすることができるようにした．

さらに, Model や View の中でも, キャンバスやパレットといった役割の違うものを分けることで, ソースコードの意味を分かりやすくした．

各クラス間のやりとりは, Observer パターンを参考に, EventListener によって行うようにした．オブジェクトの変更を Event を用いて通知することで, Listener 側が何を Listen しているか分かりやすくなった．

また, 各 Model, View, Controller のクラスは, Main クラス以外でコンストラクタを呼ばないようにした．つまり, どこかのクラスの中で, 外から見えない間に, Model が生成され, 削除されるといった状況が起こらないようにした．した．

作成したプログラムのクラス図を図 1 に示す．Window を生成する Main クラスは WizarDraw クラスである．WizarDraw クラス内で, 各 Model, View, Controller をインスタンス化して 1 つのアプリケーションとして結合している．

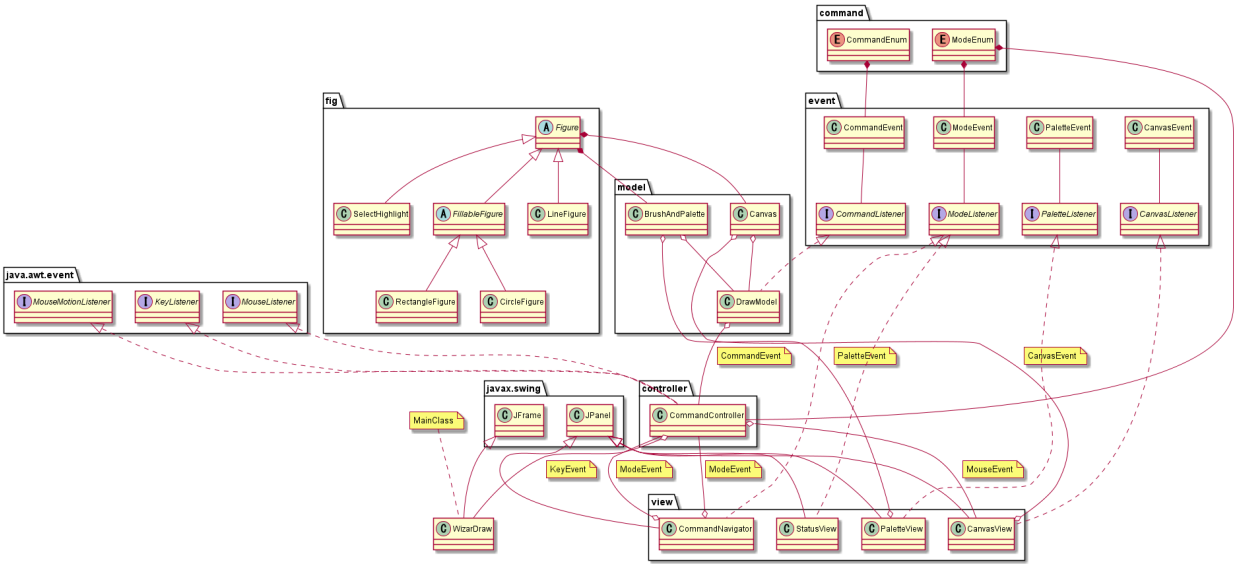

図 1　クラス概要図

2.1　コマンドシステム

コマンドシステムの動作概要を図 2 に示す．

今回作成したプログラムでは, Model に対するあらゆる操作をコマンドとして実行する. まず, ユーザーがアプリケーションに対して行った, キー入力, マウス入力を Controller が受け取る. 入力をもとに, Controller はコマンド (CommandEnum で表現される) を作成する. コマンドは CommandEvent として DrawModel に通知され, Controller クラスの内部で DrawModel を操作することはない.

コマンドを受けとった DrawModel は, BrushAndPalette や Canvas のメソッドを呼び出しコマンドを実行する. コマンドの結果, Model に変更があれば, それぞれ View に Event が発行される.

このようなコマンドシステムの実装によって, Model, View, Controller が明確に分離され, UI の変更が容易になる. さらには, 作業記録をすべてテキストベースデータで保存することや, マクロ機能の実装も可能となる (今後の課題とする).

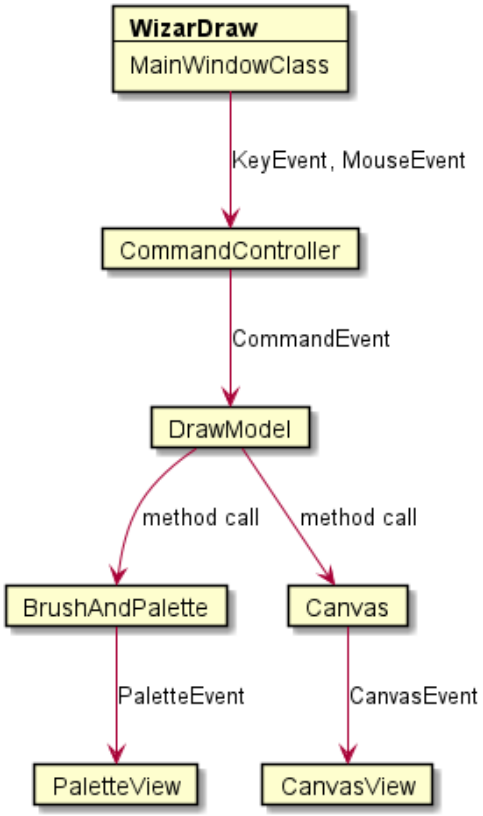

図 2　コマンドシステムの動作概要

2.2　モード

すべての操作をコマンドとして実装するのだが, 1 つのコマンドを 1 つのキーに割り当てると, キーが足りなくなる. そこで, テキストエディタの Vim にならって, モードを導入した.

コマンドを入れる前に, ユーザに適切なモードに切り替えてもらう. 色に関する操作をしたいときに, c キーを押して, Color モードに入るといった感じである.

モードは, View と Controller にのみ導入し, Model には導入しなかった. その理由を以下で説明する.

ユーザーの視点では, モードの切替もコマンドに見えるため, コマンドとモードが木構造になっ

ているように見える．よって, View や Controller 上ではコマンドとモードが木構造のように表現されている．一方で, Model ではモードが存在せず, コマンドとモードは何の構造も持っていない．これによって, UI と Model が分離され, UI の工夫の自由度が向上する．

2.3 図形の描画

図形を Model に追加して, View に反映するまでの動作を, 図 3 に示す．

まず, マウス入力から, CommandController が CreateFigureCommand を CommandEvent として, DrawModel に通知する．それを受け取った DrawModel が, BrushAndPalette オブジェクトから, 描画すべき Figure オブジェクトを取得する．その後, 描画すべき Figure オブジェクトが Canvas オブジェクトに渡される．Canvas オブジェクトは受け取った Figure を, 選択中の図形として扱い, 描画すべき Figure を格納したスタックに追加する．

これまでの処理で, Canvas オブジェクトが変更されたので, 変更が CanvasListener に通知される．今回は, CanvasView が変更を Listen して View を更新する．

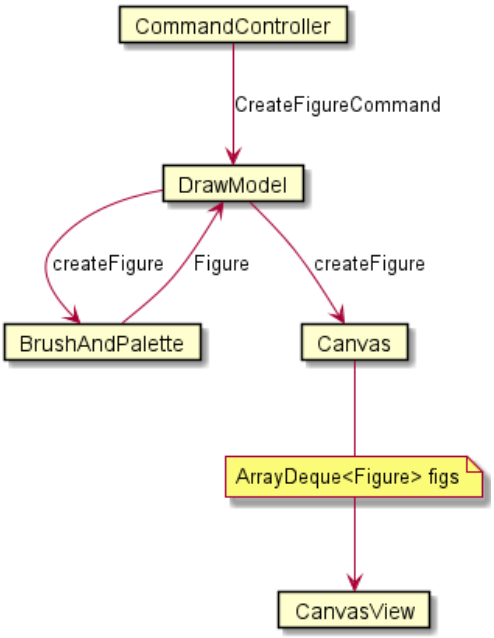

図 3　createFigure の動作概要

3 プログラムの説明

各クラスについて説明する．プログラムは, 計算機室の環境である Java 1.8 を使用して書いた．

3.1 WizarDraw

ソースコードをプログラム 1 に示す．

WizarDraw クラスは Main クラスであり, アプリケーションの初期化処理や, メインウィンドウの生成等を行う．初期化処理はすべてコンストラクタ内で行われ, Model, View, Controller のクラスはすべて, このコンストラクタでインスタンス化され, 結合される．

3.2 command

このパッケージは, コマンドシステムに関連する Enum をまとめたものである. 基本的に MVC の他のクラス上で, 自由に使われることを想定している. そのため, ここに宣言してある Enum が何か機能を有しているわけではない, あくまで, データを分かりやすく表現するために作成した.

Enum を用いることで, switch 分や map を用いた分岐のパターンを把握しやすくなる.

3.2.1 CommandEnum

ソースコードをプログラム 2 に示す.

CommandEnum は CommandController から DrawModel へコマンドを送信するために作成した. また, Command の一覧を UI に表示するときに使う, コマンドの名称とショートカットのテキストもここに書いた.

3.2.2 ModeEnum

ソースコードをプログラム 3 に示す.

ソフトウェアの動作モード, 及び, CommandController のモードを表すために作成した. また, UI 上で現在のモードを表示するときに使う, モードの名称と切り替えキーもここに書いた.

3.3 controller.CommandController

ソースコードをプログラム 4 に示す.

CommandController は Mouse と Key の Event を Listen して, DrawModel にコマンドを実行させる. キー入力は processCommand 関数で処理される. 現在の CommandController の mode に応じて, 各モードの Processor を呼び出し, それぞれその中でキー入力を判別して, コマンドを実行する.

コマンドは CommandListener の commandPerform に CommandEvent を渡して実行される. そのとき, コマンドによって, 追加でマウスの座標や, 色変更の関数を指定する必要がある. 追加の引数がいらないコマンド実行する simpleCommandPerform を定義することで, 単純なコマンドの呼び出しを簡略化した.

また, CommandController は View に対して, 現在のモードを知らせる必要があるので, modeChange が呼び出される度に, ModeListener に対して Event を送出している.

3.4 model

プログラムの中でも, UI と関係しない, ソフトウェアのロジックそのものを model として分離した.

3.4.1 DrawModel

ソースコードをプログラム 21 に示す.

DrawModel の役割は, CommandEvent を Listen して, 適切にコマンドを実行することである. DrawModel の役割はコマンドの解釈なので, 実際の model としての仕事は, PaletteAndBrush オ

ブジェクトや Canvas オブジェクトが行う.

コマンドの処理部分は, CommandEnum を switch 文で処理する方式ではなく, HashMap を用いた分岐方式にした. これによって, 分岐処理の自由度が下がり, 各コマンドの独立性が上がる. また, すべてのパターンの処理が実装されていることをテストコードで検証しやすくなる (今回は時間の都合上テストコードは未実装).

3.4.2 PaletteAndBrush

ソースコードをプログラム 22 に示す.

PaletteAndBrush の役割は 3 つのカラーパレットと, 次に描画する図形の種類を保持することである.

model を Canvas と分離することで, View に対して, パレットの変化だけを伝えられるようにした.

3.4.3 Canvas

ソースコードをプログラム 20 に示す.

Canvas の役割は描画すべき図形を保持すること, それらを操作することである.

canvasFigs は描画される図形, selectedFigure が現在操作対象の図形, highlightFigure は選択中の図形を分かりやすくするために表示する図形である.

また, Undo・Redo コマンドを実装するために, delete された図形はすべて, deleteFigs というスタックに格納される.

select 関数では図形の選択を実装している. この関数は, 現在選択中の図形よりも下にある図形を選択するように実装されている.

3.5 view

3.5.1 CanvasView

CanvasView はウィンドウの右下に表示される要素である.

ソースコードをプログラム 23 に示す.

CanvasView の役割は Canvas オブジェクトが変更されるたびに, その中身をウィンドウに反映することである. 変更の検知を Listner で実装することはできたが, Canvas の内容を取得する部分が完全に分離しきれなかった.

3.5.2 PaletteView

PaletteView はウィンドウの左上に表示される要素である.

ソースコードをプログラム 25 に示す.

PaletteView の役割は PaletteAndBrush オブジェクトが変更されるたびに, その中身をウィンドウに反映することである. 変更の検知を Listner で実装することはできたが, PaletteAndBrush の内容を取得する部分が完全に分離しきれなかった.

3.5.3 CommandNavigator

CommandNavigator はウィンドウの左に表示される要素である.

ソースコードをプログラム 24 に示す.

CommandNavigator の役割は, CommandController の mode を示すことと, コマンドのチートシートを表示することである.

各コマンド要素は Item クラス, 各要素に余白を設ける Padding クラス, 黒枠で囲まれたひとかたまりの要素を表現する Div クラスを使って, View を構築した. クラスを入れ子にして, View を構築する方法は, 趣味で Web サイトを作るときに使用した ReactJS を参考にしている.

3.5.4 StatusView

StatusView は PaletteView の右側に表示される要素である.

ソースコードをプログラム 26 に示す.

StatusView の役割は, CommandController の現在の mode を表示することである. 前述の CommandNavigator を実装するまでの間使用するために実装した. 今後, 実行されたコマンドの履歴等も表示するようにしたい.

3.6 event

3.6.1 CommandEvent

ソースコードをプログラム 7 に示す.

CommandEvent はコマンドの内容を適切に CommandListener に渡すためのクラスである.

コマンドを指定する CommandEnum と, CreateFigure や ReshapeFigure コマンドで使用する座標, ColorChange コマンドで使用する関数などを保持する. validCommand 関数によって, コマンドの実行に必要な情報がすべて渡されているかどうかチェックされる.

3.6.2 CommandListener

ソースコードをプログラム 8 に示す.

CommandController でコマンドが実行されると, CommandListener の commandPerformed 関数が実行される.

3.6.3 ModeEvent

ソースコードをプログラム 9 に示す.

ModeEvent はコマンドの内容を適切に ModeListener に渡すためのクラスである.

3.6.4 ModeListener

ソースコードをプログラム 10 に示す.

CommandController の mode が変更されると, ModeListener の modeChangeed 関数が実行される.

3.6.5 CanvasEvent

ソースコードをプログラム 5 に示す.

CanvasListener のために実装したが, 現在の実装では役割を持たず, 形式的なものである.

3.6.6 CanvasListener

ソースコードをプログラム 6 に示す.

Canvas が変更されると, CanvasListener の canvasUpdated 関数が実行される.

3.6.7 PaletteEvent

ソースコードをプログラム 11 に示す.

PaletteListener のために実装したが, 現在の実装では役割を持たず, 形式的なものである.

3.6.8 PaletteListener

ソースコードをプログラム 12 に示す.

PaletteAndBrush が変更されると, PaletteListener の paletteUpdated 関数が実行される.

3.6.9 WizarDrawEventMulticaster

ソースコードをプログラム 13 に示す.

WizarDrawEventMulticaster は, 実装したすべての独自 Listener に対して, 複数の Listener をまとめあげるために実装したクラスである. LinkedList のように内部に Listener を持つことで, Event が起こったときに呼ばれる関数を再帰的に実行して, 複数の Listener に対して, Event を実行することができる.

実装は, java.awt.AWTEventMulticaster をベースに, 必要な機能に絞って実装した.

3.7 fig

3.7.1 Figure

ソースコードをプログラム 15 に示す.

すべての描画可能な図形の抽象クラスである. Figure の役割は, すべての図形が, 位置と大きさと色を持っていることを規定すること, それから, 共通の描画関数 draw を持つこと, Cloneable であることである.

基本的に, Figure の座標 (x, y) は図形の左上の座標, 大きさ (width, height) は正の数である.

3.7.2 FillableFigure

ソースコードをプログラム 16 に示す.

Figure の中でも, 塗りつぶしが可能なことを規定する抽象クラスである. Figure に isFilled というメンバ変数が追加されている.

3.7.3 CircleFigure

ソースコードをプログラム 14 に示す.

CircleFigure は枠線のみの楕円と, 塗りつぶされた楕円を描画する Figure である.

3.7.4 RectangleFigure

ソースコードをプログラム 18 に示す.

RectangleFigure は枠線のみの四角形と, 塗りつぶされた四角形を描画する Figure である.

3.7.5 LineFigure

ソースコードをプログラム 17 に示す.

LineFigure は太さ 1 の線分を描画する Figure である. draw 関数内で呼び出している, drawLine の仕様上, LineFigure の位置は左上の座標ではなく, 線分の始点を保持しなければいけない. これはもともとの Figure の仕様と違うので, もとの Figure の仕様と合わせるために, getter で少し処理を行っている.

3.7.6 SelectHighlight

ソースコードをプログラム 19 に示す.

SelectHighlight は現在選択中の Figure を分かりやすくするために, 上に透かして表示するための図形である.

4 実行例

4.1 Normal モード

ソフトウェアを起動したとき, デフォルトで Normal モードに入る. 起動した直後のウィンドウを図 4 に示す.

図 4 起動直後のウィンドウ

このモードでは, 図形の描画とモードの切り替えができる. 左側の赤い枠は現在のモードを示している. 例えば, s キーを押して Select モードに移行すると, 赤い枠は図 5 のように変化する.

4.2 File モード

File モードでは, q キーを押すとソフトウェアを終了できる.

図 5　Select モードのウィンドウ

4.3　Brush モード・Color モード

Brush モード・Color モードでは, それぞれ, 描画する図形の変更と, その図形の色の変更ができる.

現在実装済みの図形を, すべて使って描画をした様子を図 6 に示す.

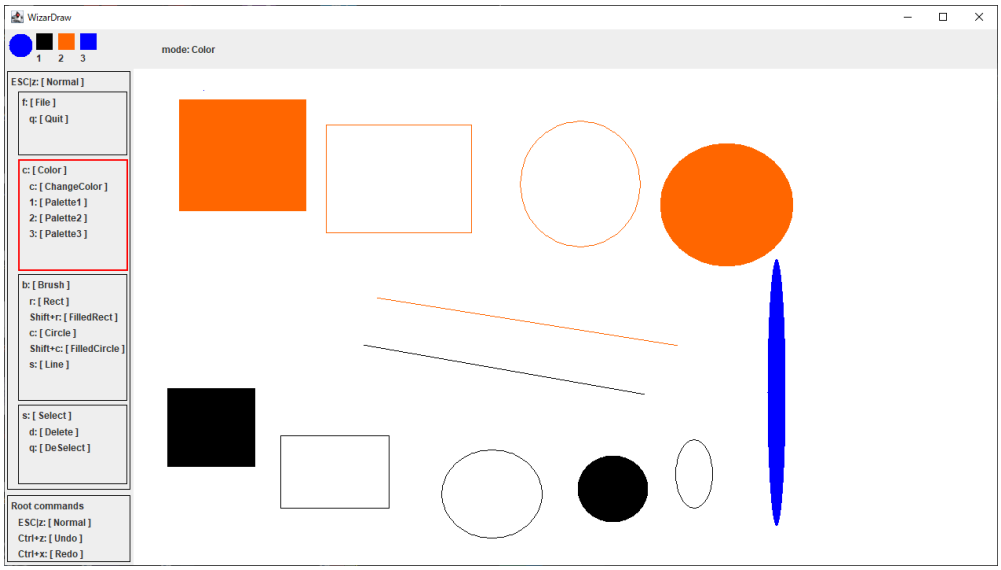

図 6　実装済み図形をすべて使って描画した様子

また, 次に描画する図形とその色は左上の PaletteView で示される.

図形に塗りつぶされた円, 色に 1 番目のパレット (赤色を保持) を選択した様子を, 図 7 に示す. Normal モードから, b キー, Shift+c キー, ESC キー, c キー, 1 キーを順に押すと, この状態になる.

また, Color モードで c キーを押すと, 現在選択中のパレットの色を変更するダイアログが出現

図 7　塗りつぶされた円と 1 番目のパレットが選択された状態

する. その様子を図 8 に示す.

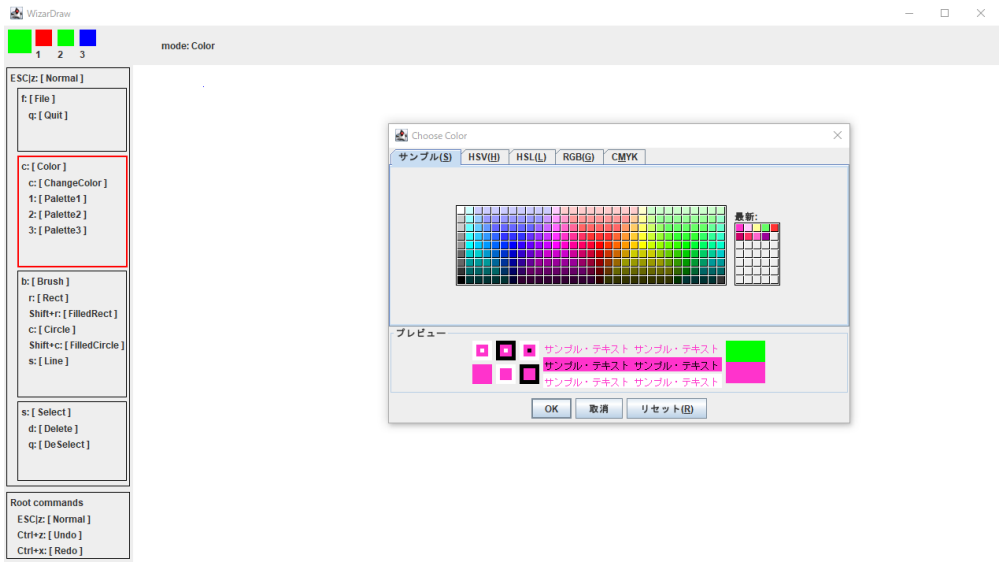

図 8　パレットの色を変更する様子

4.4　Select モード

Select モードでは, クリックした図形を選択することができる. 図形が選択された様子を図 9 に示す. d キーを押すことで, 選択された図形を削除することができる.

また, 選択中の図形をもう一度クリックすることで, その図形の下にある図形を選択することができる.

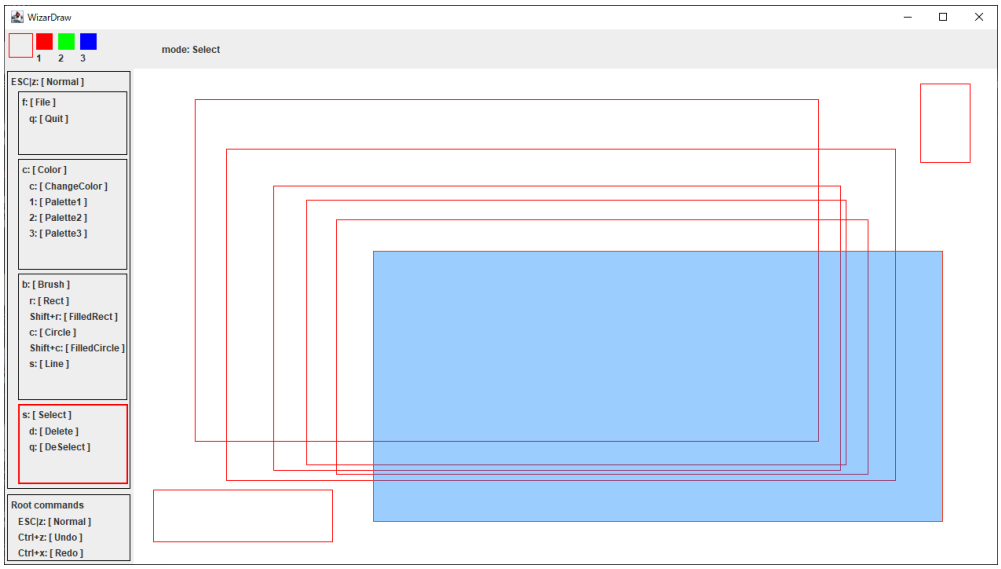

図 9　パレットの色を変更する様子

4.4.1　Undo・Redo コマンド

Undo・Redo コマンドはどのモードでも使えるコマンドである．このコマンドは一般的な Undo・Redo ではない．Undo は描画領域で 1 番上にある図形をゴミ箱スタックに移動させるもので，Redo はゴミ箱スタックの 1 番上の図形を描画領域に戻すコマンドである．

図 9 から Undo コマンドを複数回実行した様子を図 10 に示す．

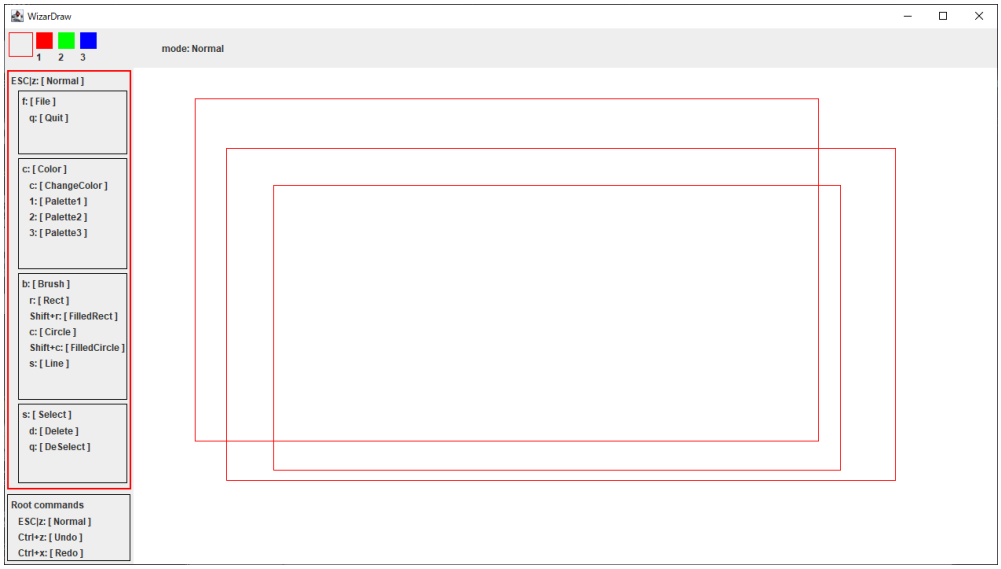

図 10　図 9 から Undo した様子

5　考察

5.1　UI について

当初の目標は以下の 2 点であった．

- マウスはキャンバス上から極力動かさない.
- 左手はキーボードのホームポジションから極力動かさない.

一般的なドローエディタがマウスでクリックするボタンで実現している機能を, キー入力で実現することによって, 目標は達成できた.

さらに, コマンドを覚えないといけないという, キー操作の欠点を, 分かりやすいチートシートをウィンドウに組み込むことで改善した.

5.2 ソフトウェアの設計について

コマンドシステムによって, Model, View, Controller の分離を達成した.

最小限の雛形となるプログラムから, 新しい機能を追加するたびに少しずつ, 設計を改善した結果現在の形になった. 当初のコマンドシステムは, Model と Controller の分離が今のプログラム程分離されていなかった. コマンドシステムのモードを, Model が持つか Controller が持つか決まっていなかった. 1 つづつ機能を実装していくにつれて, モードを Model から削除し, Model と Controller を完全に分離するほうが, クラスの役割がはっきりすることが判明した. それによって, ユーザから見ると木構造を持っているコマンドシステムを, プログラム上で平らでシンプルな構造で表現できた.

このように, 分離度の高いプログラムを作成したため, 今後新機能を実装するとき, どのクラスを変更すればよいか明確になった. 新しいコマンドを実装するには, 以下の手順で変更を加えればよい.

1. CommandEnum に追加する.
2. CommandController の適切な場所で commandPerform を呼ぶ.
3. DrawModel に新しいコマンドを処理する文を追加する.
4. 適宜新しいコマンドに対応するために, Canvas や PaletteAndBrush に関数を追加する.
5. 新しいコマンドが増えたことを分かりやすくするため, CommandNavigator に表示を追加する.

新しいコマンドがどのような機能であれ, 大まかには, このような手順で変更すればよいことが分かる.

欠点として, コマンドシステムがない場合に比べて, 手順が多くなるかもしれない. しかし, 新しいコマンドが上手く動かなかったとき, これらの手順の途中でバグが起きたことが明らかなため, デバッグがしやすい.

5.3 今後のアップデートについて

- document comment を完成させる.
  メンテナンス性の高いコードを目指すために, javadoc の作成を試みた. しかし, 時間が足りず, 一部のクラスのみ document がある状態になった. OpenSource にすることも視野に入れて, 完成を目指したい.
- テストコードを書く.
  テストコードを書きやすくするために実装した部分もあったが, 時間が足りずテストコード

が書けなかった. Java を使った開発がはじめてだったため, テストコードの書き方を習得するコストが大きかったからである. 後述の追加機能を実装する前に, まずテストコードを追加したい. 具体的には Model 周りや, Command を処理する部分が全パターン網羅してるかなどをテストしたい.

- 保存機能を実装する.
  ドローエディタの基本機能の中で, 保存機能の実装が間に合わなかった. 新機能として, まずこれを実装したい.
- マクロ機能を実装する.
  今回実装したコマンド機能は, ドローエディタに対するあらゆる操作をテキストベースで表現可能にできる. そのため, より効率的なドローエディタを目指して, マクロ機能を実装したい.

6 感想・反省

Java をきちんと書いたことは初めてだったが, 癖が少ない言語だと思った. 趣味で Rust という言語を触っていたため, Rust の機能や関数型言語風の機能を使いたくなったことが何度があった. しかし, Python や Javascript と違って型があり, 標準ライブラリの挙動も分かりやすかったので, 言語仕様でつまずくことはなかった.

反省は, やりたいことを挙げすぎたのと, リファクタリングに時間をかけすぎたために, 他の講義のための時間を圧迫してしまったことである. 今後作るときはもっと, 最初に, 最小で簡単な仕組みを作ることを意識したいと思った.

付録 A　操作法マニュアル

A.1　What is *WizarDraw*

WizarDraw は, Vim に着想を得た効率を求める人のための, 図形描画ソフトです. シンプルで学びやすいコマンド機能によって, 煩わしいボタンクリックをすべてなくしました.

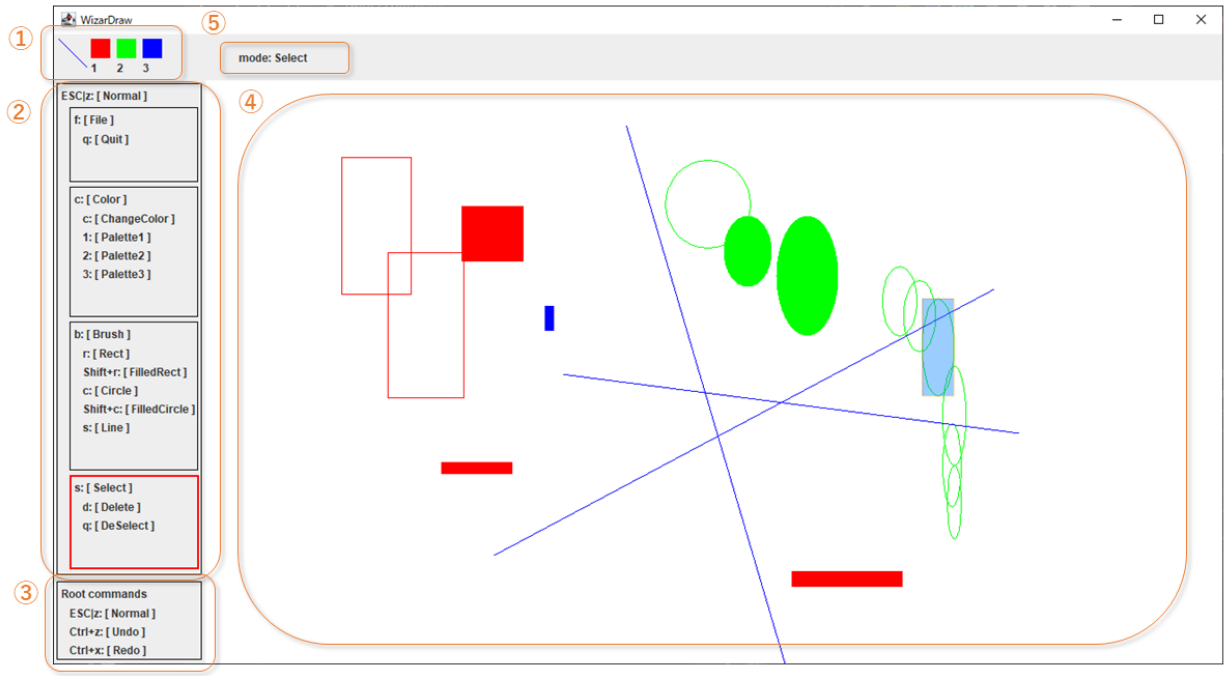

図 11　WizarDraw の Window

A.2　画面説明

① ブラシとパレット
1 番右がブラシの状態, 番号が振られた四角が現在のパレットの色を表している.
パレットの番号は, 後述のコマンド機能で切り替える番号.

② コマンド一覧
実装されてあるコマンドと現在のモードを示している.
赤枠で囲まれた部分が現在のモード, 枠内のインデントされた部分のコマンドが使える.
コマンドシステムの詳細は後述.

③ ルートコマンド一覧
どのモードでも使えるコマンドの一覧.

④ キャンバス
マウスを使って図形を描画する領域.

⑤ ステータス表示
現在のモードを表示している.

A.3 コマンドシステム

WizarDraw では, ボタンをクリックして図形を切り替えるなどの操作が存在しません. 図形の切り替え等はキーボードから特定のキーを入力することで行います. コマンドに使うキーなどは, 画面左側のコマンド一覧で常に見ることができます.

コマンドを扱うためには, モードを理解する必要があります. まず, 1 番基本となる Normal モードから説明していきます.

A.3.1 Normal モード

起動時に最初に入るモードです. このモードはモードを切り替えるために存在しているのでこれといって機能を持ちません. モードを切り替える際に必ず通過する交差点のようなものです.

どのモードにいるときでも, ESC キー, または, z キーを押すことで Normal モードに入れます.

A.3.2 File モード

Normal モードで, f キーを押すことで入れます. q キーを押すことでソフトを終了できます.

A.3.3 Color モード

Normal モードで, c キーを押すことで入れます. 1,2,3 キーを押すことで, 次の描画する図形を, その番号のパレットの色に変えられます. また, c キーを押すことで現在指定しているパレットの色を ColorChooser ダイアログを用いて変更できます.

A.3.4 Brush モード

Normal モードで, b キーを押すことで入れます. r で四角の枠線, Shift+r で塗りつぶされた四角, c で楕円の枠線, Shift+c で塗りつぶされた楕円, s で直線を描画できます.

A.3.5 Select モード

Normal モードで, s キーを押すことで入れます. このモードでは新たに図形を描画できません. 選択されている図形が青くなるので, その状態で d キーを押すことで削除できます. また, q キーで選択解除できます. 複数回クリックすることで, 今選択されている図形よりも下にある図形も選択できます.

A.3.6 Undo・Redo コマンド

どのモードにいても使えるコマンドです. Ctrl+z で Undo, Ctrl+x で Redo できます. ただし, 一般的な Undo・Redo ではなく, ただたんに描画されている図形を上から, ゴミ箱スタックに移動しているだけです.