

# Dokumentacja systemu VecEdit

Zespół projektowy:

Adam Rogowski, Szymon Sawoń, Bartosz Siemaszkiewicz

13 stycznia 2025

## Spis treści

<b>1</b>	<b>Wstęp</b>	<b>1</b>
<b>2</b>	<b>Wzorce projektowe i ich zastosowanie</b>	<b>2</b>
2.1	Kompozyt ( <i>Composite</i> ) . . . . .	2
2.2	Metoda wytwórcza ( <i>Factory Method</i> ) . . . . .	2
2.3	Strategia ( <i>Strategy</i> ) . . . . .	3
2.4	Adapter . . . . .	3
2.5	Odwiedzający ( <i>Visitor</i> ) . . . . .	4
2.6	Prototyp ( <i>Prototype</i> ) . . . . .	4
<b>3</b>	<b>Opis użytych bibliotek i narzędzi</b>	<b>4</b>
3.1	Raylib (5.5) . . . . .	4
3.2	RayGui . . . . .	5
3.3	C++20 i CMake . . . . .	5
<b>4</b>	<b>Instrukcja użytkownika</b>	<b>5</b>
4.1	Uruchomienie programu . . . . .	5
4.2	Interfejs programu — krótki opis . . . . .	5
<b>5</b>	<b>Instrukcja instalacji i kompilacji</b>	<b>7</b>
<b>6</b>	<b>Podział pracy w zespole</b>	<b>7</b>
<b>7</b>	<b>Podsumowanie</b>	<b>8</b>

## 1 Wstęp

VecEdit to aplikacja (napisana w **C++20**) służąca do edycji prostych grafik wektorowych. Umożliwia rysowanie, modyfikowanie, grupowanie oraz klonowanie obiektów graficznych (takich jak prostokąty, okręgi czy wielokąty). Program korzysta z biblioteki **raylib** (w wersji 5.5) do wyświetlania okna, rysowania figur na ekranie oraz pobierania zdarzeń wejściowych od użytkownika (klawiatura, mysz). Do tworzenia interfejsu graficznego w trybie *immediate mode* używana jest biblioteka **RayGui**.

Aplikacja pozwala na:

- rysowanie i edycję kształtów (w tym zmianę koloru, obrysu, przezroczystości),

- przesuwanie i skalowanie figur,
- grupowanie obiektów (`FigureGroup`),
- cofanie (`undo`) i ponawianie (`redo`) operacji dzięki wzorcowi *Command*,
- zapisywanie projektów do plików wektorowych (`SVG`) i *bitmapowych* (`PNG`),
- otwieranie, zamykanie i przełączanie się między wieloma zakładkami dokumentów.

## 2 Wzorce projektowe i ich zastosowanie

Poniżej przedstawiono zastosowane wewnątrz aplikacji wzorce projektowe, z uwzględnieniem istotnych elementów w kodzie oraz komentarzy dotyczących możliwych zmian (tzw. *wektora zmian*).

### 2.1 Kompozyt (*Composite*)

- **Cel:** Umożliwienie traktowania pojedynczych obiektów (figury) i grup obiektów (grupy figur) w sposób jednolity, dzięki interfejsowi `Figure`.
- **Struktura:**
  - `figure` – przestrzeń nazw, w której znajdują się klasy komponentów
  - `Figure` – interfejs *komponentu*. Posiada m.in. metody typu `accept(visitor)`, które w implementacjach *Composite* przekazują wywołanie wszystkim dzieciom. (`src/figure/Figure.h`)
  - `FigureGroup` – *kompozyt*, zawiera wektor `children` oraz metody `addChild(...)` itp. (`src/figure/FigureGroup.h`)
  - `RectFigure`, `CircleFigure`, `PolyFigure` – *konkretne komponenty (liście)*, nie posiadają dzieci. (`src/figure/*Figure.h`)
- **Użycie:**
  - W edytorze (`Editor - src/ui/Editor.cpp`) przy `groupFigures()` i `ungroupFigures()` łączymy/rozbijamy figury w drzewiastą strukturę.
  - `accept(visitor)` w `FigureGroup` wywołuje `accept` u wszystkich dzieci, co integruje się z *Visitor*. (`np. ui::Editor::renderMainContent()`)
- **Wektor zmian:** Dodanie kolejnych typów figur nie wymaga zmian w `FigureGroup`, wystarczy zaimplementować metody interfejsu `Figure` - najlepiej dziedzicząc po `FigureGroup`.

### 2.2 Metoda wytwórcza (*Factory Method*)

- **Cel:** Abstrakcja sposobu tworzenia obiektów. W naszym projekcie służy do tworzenia odpowiedniej implementacji `PointEditor` w metodzie `Figure::makePointEditor()`.
- **Struktura i użycie:**
  - `Figure` deklaruje wirtualną metodę `makePointEditor()`, która w podklasach (`RectFigure`, `CircleFigure`, ...) zwraca odpowiedni edytor (`np. RectFigurePointEditor`).

- Editor korzysta z tej metody, by pobierać adapter do edycji punktów figury, nie wiedząc *jaki to* konkretnie edytor.
- **Wektor zmian:** Nowy typ figury (np. `StarFigure`) wystarczy wyposażyć w swoją implementację `makePointEditor()`, zwracając własną implementację `PointEditor` np. `StarFigurePointEditor`.

## 2.3 Strategia (*Strategy*)

- **Cel:** Definiowanie wymiennych algorytmów/akcji, które można przypisać do przycisków lub skrótów klawiaturowych (np. *undo*, *redo*).
- **Kluczowe elementy:**
  - `ui::strategy` – przestrzeń nazw, w której znajdują się klasy strategii
  - `Strategy<...>` – generyczny interfejs (plik `Strategy.h`) – argumenty szablonu odnoszą się do typów argumentów przekazywanych podczas wywołania strategii,
  - `UndoStrategy`, `RedoStrategy`, `SetSelectStrategy` itp. – konkretne strategie używane przez guziki i skróty klawiszowe,
  - `FunctorStrategy<...>` – pozwala zdefiniować strategię w oparciu o lambda-/domknięcie (*closure*), bez tworzenia nowej podklasy.
- **Użycie:**
  - W `AppUi` (np. `ui::AppUi::setupUndoRedoButtons()`) do `IconButton` lub `KeyboardShortcut` przypisujemy strategię. Np. `auto str = std::make_shared<UndoStrategy>` a w innym miejscu `addShortcut(str, KEY_Z, mod)`.
  - W `FunctorStrategy` wystarczy przekazać lambda, np. `FunctorStrategy<>{[this]() { editor->exportDocument("png"); }}`.
- **Wektor zmian:** Aby dodać nową prostą akcję, można utworzyć instancję `FunctorStrategy` z odpowiednim wyrażeniem lambda; dla bardziej złożonych przypadków — osobną klasę dziedziczącą po `Strategy`.

## 2.4 Adapter

- **Cel:** Udostępnienie wspólnego interfejsu edycji punktów figur (`PointEditor`) mimo iż każda figura ma odmienne właściwości i sposób ich zmiany.
- **Struktura i użycie:**
  - `figure::adapter` – przestrzeń nazw, w której znajdują się klasy adapterów
  - `PointEditor` – interfejs z metodami `getEditPoints()` i `updatePointPosition(...)`,
  - `RectFigurePointEditor`, `CircleFigurePointEditor`, `PolyFigurePointEditor` – adaptery do konkretnych figur,
  - Każdy adapter *zwraca* (np. w `getEditPoints()`) położenia uchwytów charakterystyczne dla swojej figury (np. promień dla koła).
- **Wektor zmian:** Dodanie nowego kształtu wymaga stworzenia nowego adaptera do edycji, a sama logika `Editor` (obsługująca `PointEditor`) nie musi być zmieniana.

## 2.5 Odwiedzający (*Visitor*)

- **Cel:** Oddzielenie logiki przetwarzania figur (np. rysowania, zapisu do pliku) od samych klas figur.
- **Struktura i użycie:**
  - `figure::visitor` – przestrzeń nazw, w której znajdują się klasy odwiedzających
  - `FigureVisitor` – interfejs odwiedzającego (`src/figure/visitor/FigureVisitor.h`),
  - `RendererVisitor` (renderuje figury na ekranie) (`src/figure/visitor/RendererVisitor.cpp`),
  - `SvgSerializerVisitor` (zapisuje wektorowo do pliku SVG),
  - `BitmapRendererVisitor` (obsługa zapisu do formatów bitmapowych np. PNG).
  - Metoda `accept(visitor)` w `Figure` (bądź `FigureGroup`) wywołuje `visitor.visit(...)` odpowiedniego typu.
- **Wektor zmian:** Dodanie kolejnej operacji (np. liczenie pola figur) wymaga utworzenia nowej klasy odwiedzającego, bez modyfikacji istniejących figur. Z kolei dodanie nowej figury pociąga za sobą modyfikację wszystkich odwiedzających tak by obsłużyć odwiedzenie nowego rodzaju figury.

## 2.6 Prototyp (*Prototype*)

- **Cel:** Możliwość tworzenia nowych instancji obiektów (figur) poprzez klonowanie bez tworzenia zależności pomiędzy kodem wywołującym a konkretną klasą, której obiekt jest tworzony.
- **Struktura i użycie:**
  - `FigureBase<FigureType>` posiada wirtualną metodę `clone()`, a w podklasach (`RectFigure`, `CircleFigure` itp.) implementuje tworzenie kopii obiektu *danego* typu.
  - W `Editor::processModeInsert()` tworzymy nową figurę poprzez sklonowanie istniejącego prototypu (figura-prototyp jest wcześniej ustawiona przez `SetFigureInsertStrategy`).
- **Wektor zmian:** Każda klasa dziedzicząca z `FigureBase<FigureType>` automatycznie dziedziczy implementację operacji klonowania (`src/figure/FigureBase.h: figure::FigureBase<FigureType>::clone()`) pozwalającej na tworzenie nowych instancji typu `FigureType`.

## 3 Opis użytych bibliotek i narzędzi

### 3.1 Raylib (5.5)

W projekcie korzystamy z **Raylib** (<https://www.raylib.com/>) w wersji 5.5, która zapewnia:

- Wyświetlanie okna aplikacji w trybie 2D (np. `InitWindow`, `WindowShouldClose`),

- Funkcje do rysowania podstawowych kształtów (np. `DrawRectangle`, `DrawCircle`),
- Obsługę zdarzeń wejściowych (np. kliknięcia myszą (`IsMouseButtonPressed`), klawisze (`IsKeyDown`)),
- Zapisywanie wyrenderowanej bitmapy do pliku (np. `ExportImage`).

## 3.2 RayGui

Używamy **RayGui** (<https://github.com/raysan5/raygui>) – dodatek do Raylib, która wspiera tworzenie interfejsu graficznego w tzw. *immediate mode* — przyciski, suwaki i teksty generowane są bezpośrednio podczas rysowania każdej klatki i natychmiastowo obsługiwana jest ich interakcja. Umożliwia to proste tworzenie interfejsów (dodanie jednej kontrolki to wywołanie jednej funkcji np. `GuiButton`) użytkownika jednocześnie pozwalając nam stworzyć własny szkielet wzorców projektowych obsługujących ten interfejs.

## 3.3 C++20 i CMake

- **C++20** — W projekcie używamy `std::filesystem` do obsługi plików, `std::format` do wygodnego formatowania tekstu, `std::ranges` to operacji na kolekcjach. Wspierane są kompilatory GCC (14.2.1) i clang (18.1.8).
- **CMake 3.29** — Plik `CMakeLists.txt` konfiguruje system budowania i podczas generowania konfiguracji projektu (np. dla GNU Make lub Ninja) automatycznie pobiera Raylib z GitHuba.

# 4 Instrukcja użytkownika

## 4.1 Uruchomienie programu

VecEdit można uruchomić:

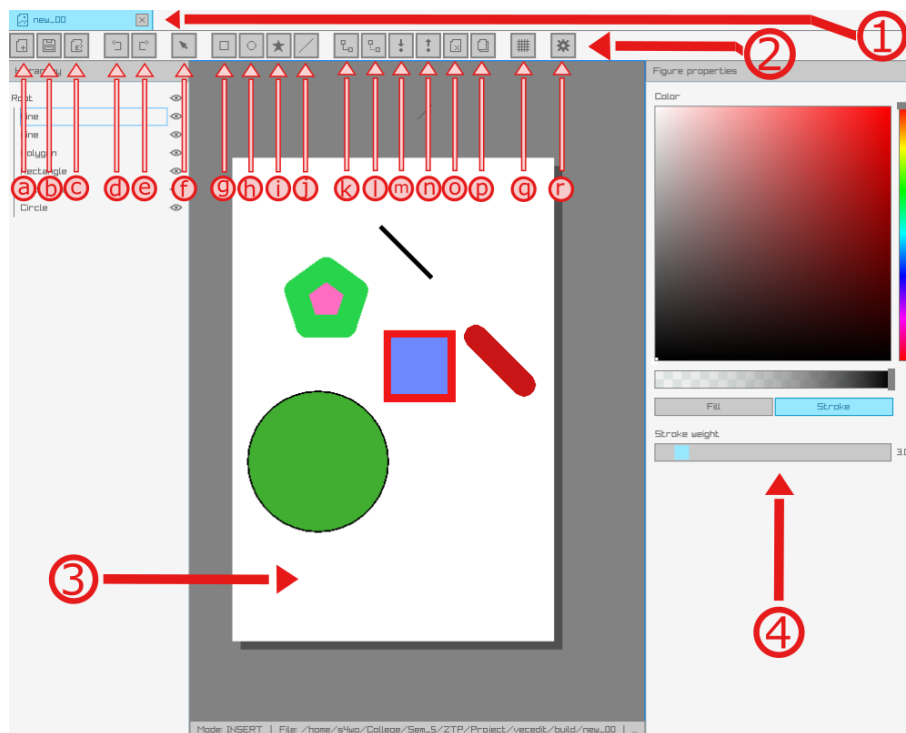
- **Z gotowego pliku wykonywalnego:** na platformie Windows lub Linux dostarczamy *executable* w folderze `bin/`.
- **Z linii poleceń po kompilacji:** np.

```
cd build
./VecEdit
```

## 4.2 Interfejs programu — krótki opis

Oznaczenia na zrzucie:

1. **Pasek zakładek dokumentów** — wybór otwartego projektu, możliwość zamknięcia karty.
2. **Pasek narzędzi** (z przyciskami od **A** do **S**):
  - a) **New** (skrót `Ctrl+N`) — tworzy nowy dokument.
  - b) **Save** (skrót `Ctrl+S`) — zapisuje obecny dokument do SVG.



Rysunek 1: Przykładowy widok programu.

- c) **Export** (skrót **Ctrl+E**) — eksport do PNG.
  - d) **Undo** (skrót **Ctrl+Z**) - cofnięcie zmian
  - e) **Redo** (skrót **Ctrl+Y**) - przywrócenie zmian
  - f) **Select** (skrót **Q**) — narzędzie zaznaczania figur.
  - g) **Insert Rectangle** (skrót **R**) — wstawianie prostokąta.
  - h) **Insert Circle** (skrót **E**) — wstawianie koła.
  - i) **Insert Polygon** (skrót **T**) — wstawianie wielokąta gwiazdy.
  - j) **Insert Line** (skrót **L**) — wstawianie linii.
  - k) **Group** (skrót **Ctrl+G**) - grupowanie zaznaczonych figury obiektów.
  - l) **Ungroup** (skrót **Ctrl+U**) - odgrupowanie obiektów.
  - m) **Move Lower** (skrót **PageDown**) - przeniesienie zaznaczonej figury w dół w hierarchii obiektów.
  - n) **Move Higher** (skrót **PageUp**) - przeniesienie zaznaczonej figury w górę w hierarchii obiektów.
  - o) **Delete Figure** (skrót **Delete**) - usunięcie zaznaczonej figury.
  - p) **Duplicate Figure** (**Ctrl+D**) - zduplikowanie figury.
  - q) **Toggle Grid** (**Ctrl+I**) — włączenie/wyłączenie przyciągania do siatki.
  - r) **Document Properties** (**Ctrl+.**) — otwiera panel właściwości dokumentu (nazwa pliku i rozmiar płótna).
3. **Obszar roboczy** — kliknięcie w tym obszarze lewym przyciskiem myszy wstawia figurę (jeśli wybrano narzędzie *Insert*) lub zaznacza figurę (jeśli wybrano narzędzie

*Select* - trzymając Shift można zaznaczać wiele figur). Koło myszy pozwala na przybliżanie i oddalanie widoku jak również podczas trzymania Shift na przesuwanie widoku.

4. **Panel właściwości** (z boku lub w oknie) — służy do zmiany kolorów, grubości obrysu i przezroczystości dla wybranej figury.

## 5 Instrukcja instalacji i kompilacji

Kompilacja ze źródeł odbywa się w następujących krokach:

1. **Rozpakowanie zip** - źródła projektu dostarczone są w archiwów zip, które należy rozpakować. **TODO: dodać dokładną nazwę folderu ze źródłami**

2. **Kompilacja z użyciem CMake**

- **W linii poleceń** - z wykorzystaniem GNU Make (proces opisany również w README.md)

```
mkdir build
cd build
cmake .. # ten krok może długo potrwać, ponieważ pobiera Raylib
make
```

- **Kompilacja z użyciem środowiska CLion:**

- (a) Po zaimportowaniu projektu wykonać File->Reload CMake Project
- (b) Następnie po ładowaniu można skompilować i uruchomić projekt za pomocą przycisku "Run" lub SHIFT+F10
- (c) Dodatkowo w folderze `vecedit/cmake-build-debug` jest plik `.exe`.

3. **Uruchomienie:** Po kompilacji w folderze `build` znajduje się plik `VecEdit` (Linux/macOS) lub `VecEdit.exe` (Windows). Można go uruchomić z wiersza poleceń lub przez dwuklik.

## 6 Podział pracy w zespole

Adam Rogowski

- **Composite (FigureGroup):** stworzenie klasy grupującej figury i zarządzanie hierarchią dzieci (w tym obsługa zagnieżdżonych grup).
- **Command** (w tym `RemoveFiguresCommand`, `GroupFiguresCommand` i `CommandManager`): implementacja głównego mechanizmu `undo/redo`, obejmująca tworzenie i cofanie poleceń.
- **Zarządzanie warstwami i blokowaniem figur:** dodatkowo rozbudował `Editor` o możliwość przesuwania obiektów między warstwami oraz blokowania/odblokowywania figur w drzewie.
- **Interfejs użytkownika (AppUi, Toolbar):** stworzenie paska narzędzi (z przyciskami od `New` do `Settings`), logika rozmieszczania widgetów i współpracy z `Editor` (m.in. przełączanie trybów *insert* / *select*).

## Szymon Sawoń

- **Strategy**: implementacja akcji takich jak `SetSelectStrategy`, `OpenDocumentStrategy` oraz `FunctorStrategy` (do prostych poleceń tworzonych w oparciu o wyrażenia lambda).
- **Prototype**: w `FigureBase<>` oraz poszczególnych figurach (`RectFigure`, `CircleFigure` itp.) wdrożył metodę `clone()` pozwalającą na klonowanie obiektów bez znajomości ich typu.
- **Obsługa skrótów klawiaturowych** (`KeyboardShortcut`): mapowanie klawiszy na odpowiednie strategie (np. `Ctrl+N` na `NewDocumentStrategy`), przechowywanie listy skrótów i wywoływanie ich w pętli głównej.
- **Interfejs użytkownika** (`AppUi`, `Toolbar`): stworzenie paska narzędzi (z przyciskami od `New` do `Settings`), logika rozmieszczania widgetów i współpracy z `Editor` (m.in. przełączanie trybów *insert* / *select*).

## Bartosz Siemaszkiewicz

- **Visitor**: implementacja `RendererVisitor` (rysowanie figur), `SvgSerializerVisitor` (eksport do SVG) oraz `BitmapRendererVisitor` (eksport do PNG).
- **Prototype**: w `FigureBase<>` oraz poszczególnych figurach (`RectFigure`, `CircleFigure` itp.) wdrożył metodę `clone()` pozwalającą na klonowanie obiektów bez znajomości ich typu.
- **Obsługa zapisywania do plików** (SVG i PNG): integracja z `Editor` i `Document`, przygotowanie procesu serializacji (np. `SvgSerializerVisitor`) i eksportowania całości sceny w wybranym formacie.
- **Interfejs użytkownika** (`AppUi`, `Toolbar`): stworzenie paska narzędzi (z przyciskami od `New` do `Settings`), logika rozmieszczania widgetów i współpracy z `Editor` (m.in. przełączanie trybów *insert* / *select*).

## 7 Podsumowanie

`VecEdit` stanowi przykład aplikacji, w której zastosowanie wielu wzorców projektowych (kompozyt, metoda wytwórcza, strategia, adapter, odwiedzający, prototyp i command) umożliwia elastyczną i czytelną strukturę kodu:

- **Composite** — wspólna obsługa figur i grup (hierarchii),
- **Factory Method** — tworzenie edytorów punktów figur,
- **Strategy** — łatwe definiowanie akcji dla GUI (także w postaci `FunctorStrategy`),
- **Adapter** — jednolite API do edycji różnych typów figur,
- **Visitor** — rozdzielenie logiki rysowania, serializacji i bitmap,
- **Prototype** — proste klonowanie figur,
- **Command** — mechanizm cofania i ponawiania operacji.



Dzięki temu kod można łatwo rozwijać (dodawać nowe rodzaje figur, strategie, odwiedzających) i utrzymywać.

**Miłego korzystania z VecEdit!**