SWO3	**Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3**	**WS 2018/19, Angabe 10**

☐ Gruppe 1 (J. Heinzelreiter)

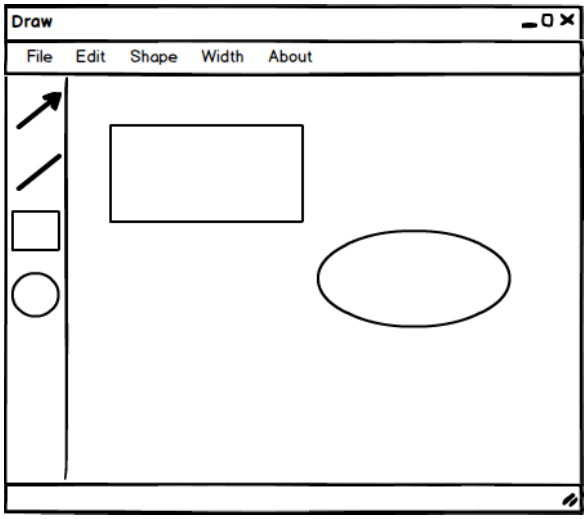
☐ Gruppe 2 (M. Hava) Name: Niklas Vest Aufwand [h]: 12

☒ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (20 P + 40 P + 40 P)	90%	100%	85%

Beispiel 1: Grafikeditor (`src/draw/`)

Implementieren Sie (basierend auf der MiniLib) einen Grafikeditor `Draw`. Mit diesem Grafikeditor kann man Linien, Ellipsen und Rechtecke zeichnen. Dazu muss man über ein Menü oder über eine Randleiste (siehe Bild unten) auswählen können, ob man zeichnen will und wenn ja, was man zeichnen will. Der Grafikeditor muss etwa folgendes Aussehen haben (hier mit einer Randleiste zur Werkzeugauswahl, der Pfeil in der Randleiste ist für die Selektion von gezeichneten Objekten vorgesehen):



(a) Implementieren Sie alle notwendigen Klassen für die Grundfunktionalität des Grafikeditors. Achten Sie beim Entwurf auf eine einfache Erweiterbarkeit für zukünftige grafische Objekte. Durch Klicken und Ziehen mit der Maus muss es möglich sein, Objekte der gerade ausgewählten Art zu zeichnen.

(b) Implementieren Sie die Funktionalität der Randleiste.

(c) Implementieren Sie die Funktionalität „Selektion und Manipulation eines Objekts“.

Beachten Sie die folgenden Hinweise:

- Testen Sie Ihr Programm, indem Sie sich ausgeben lassen, welches grafische Objekt gerade ausgewählt ist. Das ausgewählte Objekt muss in der Randleiste durch inverse Darstellung hervorgehoben werden.
- Wenn das ausgewählte grafische Objekt durch den Menüpunkt `Shape` (er hat die gleiche Funktionalität wie die Randleiste) geändert wird, dann muss sich natürlich auch das inverse Objekt in der Randleiste ändern. Durch Klicken in die Randleiste muss sich das angewählte grafische Objekt auch ändern lassen.
- Bei der Selektion eines Objekts müssen die notwendigen Selection-Marker angezeigt (und wieder gelöscht) werden. Außerdem müssen auf das ausgewählte Objekt Aktionen wie `Cut`, `Copy` und `Paste` angewandt werden können.

Ausarbeitung 10

Niklas Vest

January 27, 2019

1 Draw

1.1 Lösungsidee

a

Die Klassen für die zum Zeichnen verfügbaren Formen, sowie der "Zeichenmechanismus" wurden quasi gänzlich in der Übung implementiert. Jede einzelne Kindklasse von *shape* überschreibt die Funktion *on_draw* um eine grafische Representation von sich selbst auf den übergebenen wxWidgets Device Context zu zeichnen.

b

Die Randleiste habe ich in den Klasse *tool_bar* und *tool_bar_item* realisiert. Eine *tool_bar* kann mehrere items aufnehmen, die aus einer grafischen Representation und einem event handler bestehen. Wird ein icon geclickt, wird das damit assoziierte callback aufgerufen. Es gibt zu jedem Zeitpunkt ein "ausgewähltes" Tool, welches mittels aufruf zu *draw_inverse* "invers" gezeichnet wird. Der Pfeil und die Linie werden lediglich fetter gezeichnet.

c

Wird im Selektionsmodus in die Zeichenfläche gelickt, so werden alle Formen (visuell) von oben nach unten durchsucht. Befindet sich der Mauszeiger dabei innerhalb der Bounding Box einer Form, wird diese Objekt als "selektiert" markiert. Gleichzeitig kann man auch die Maus gedrückt halten und bewegen, um die Form zu bewegen.

Copy, Cut und Paste

Das *draw_window* hat ein internes Clipboard, einen schlichten *std::unique_ptr<shape>*. Das Fenster reagiert auf die klassischen Tastatureingaben:

- CTRL+C kopiert eine Form in das Clipboard.

- CTRL+X schneidet die Form aus dem canvas und legt sie ins Clipboard.
- CTRL+V kopiert die Form im Clipboard und fügt sie ins Canvas ein.

Dies wurde mittels der dynamisch gebundenen Methode *clone* erreicht, welche in *shape* "pure virtual" ist.

1.2 Implementierung

Listing 1: draw.cpp

```cpp
#include "draw_application.h"

int main(int argc, char *argv[])
{
    draw_application {}.run(argc, argv);
}
```

Listing 2: tool_bar.h

```cpp
#pragma once

#include <wx/dc.h>
#include <functional>
#include <memory>
#include "shape.h"

class draw_window;

/**
 * A simple callback for event handling.
 */
using event_handler = std::function <void()>;

/**
 * A clickable item displayed within a tool bar.
 */
struct tool_bar_item
{
    /**
     * The name of the goofball.
     */
    std::string name;

    /**
     * The visual representation of the item.
     */
    std::shared_ptr <shape> icon;

    /**
     * The function which is invoked upon clicking
     * the icon of the item.
     */
    event_handler callback;
};

```

```
37 class tool_bar
38 {
39 public: // methods
40     /**
41      * @param parent The window in which this tool bar will be mounted.
42      * @param width The desired with of the tool bar.
43      */
44     explicit tool_bar(draw_window *parent, unsigned width);
45 
46     /**
47      * Draws the toolbar
48      */
49     void draw(wxDC &context) const;
50 
51     /**
52      * Adds the supplied item to the toolbar. The later an item is
53      * added, the further down in the bar its icon will appear.
54      * @param item The item to add.
55      */
56     void add_item(tool_bar_item item);
57 
58     /**
59      * @return The metrics of the tool bar.
60      */
61     wxRect get_bounding_box() const;
62 
63     /**
64      * Processes clicks that happen within the tool bar.
65      */
66     void on_mouse_left_down(const ml5::mouse_event &event);
67 
68     /**
69      * Selects the tool bar item with the specified name.
70      */
71     void select(const std::string &name);
72 
73 private: // members
74 
75     /**
76      * The spacing between the tool bar border and
77      * the icons of its items.
78      */
79     static const unsigned PADDING = 10;
80 
81     /**
82      * @see tool_bar::tool_bar(draw_window *parent, unsigned width);
83      */
84     draw_window *const _parent;
85 
86     /**
87      * @see tool_bar::tool_bar(draw_window *parent, unsigned width);
88      */
89     unsigned _width;
90 
91     /**
92      * The list of items held by the tool bar.
93      */
```

```
94      std::vector <tool_bar_item> _items;
95
96      /**
97       * The index of the selected tool in tool_bar::_items.
98       */
99      std::size_t _selection { 0 };
100
101     /**
102      * The background color of the tool bar.
103      */
104     wxBrush _brush { wxBrush(wxColour(230, 230, 230)) };
105
106     /**
107      * The border-color and -width of the tool bar.
108      */
109     wxPen _pen { *wxBLACK_PEN };
110 };
```

Listing 3: tool_bar.cpp

```
1  #include "tool_bar.h"
2
3  #include <iostream>
4  #include "draw_window.h"
5
6  tool_bar::tool_bar(draw_window *parent, unsigned width) : _parent { parent },
       _width { width }
7  {}
8
9  void tool_bar::draw(wxDC &context) const
10 {
11     // draw the tool bar
12     context.SetPen(_pen);
13     context.SetBrush(_brush);
14     context.DrawRectangle(0, 0, _width, _parent->get_height());
15
16     // draw them icons
17     for (std::size_t i = 0; i < _items.size(); ++i) {
18         if (i != _selection) {
19             _items[i].icon->draw(context);
20         } else {
21             _items[i].icon->draw_inverse(context);
22         }
23     }
24
25 }
26
27 void tool_bar::add_item(tool_bar_item item)
28 {
29     // scale and position the icon relative to the
30     // other items in the tool bar
31     wxRect relative { PADDING, PADDING, static_cast<int>(_width - PADDING *
       2),
32                       static_cast<int>(_width - PADDING * 2) };
33     relative.y = static_cast<int>(PADDING +
34                                   (_items.size() * PADDING * 2) /* padding per
       item */ +
```

```
35                                  (_items.size() * (_width - PADDING * 2)));
      /* items */
36      item.icon->set_bounding_box(relative);
37
38      _items.push_back(item);
39  }
40
41  wxRect tool_bar::get_bounding_box() const
42  {
43      return wxRect(0, 0, _width, _parent->get_height());
44  }
45
46  void tool_bar::on_mouse_left_down(const ml5::mouse_event &event)
47  {
48      std::size_t i = 0;
49      while (i != _items.size() && !_items[i].icon->get_bounding_box().Contains
       (event.get_position())) {
50          ++i;
51      }
52      if (i != _items.size()) {
53          // an icon was clicked!
54          _selection = i;
55          _items[i].callback();
56      }
57  }
58
59  void tool_bar::select(const std::string &s)
60  {
61      std::size_t i = 0;
62      while (i != _items.size() && s != _items[i].name) {
63          ++i;
64      }
65      if (i != _items.size()) {
66          _selection = i;
67          _items[i].callback();
68      }
69  }
```

Listing 4: draw_window.h

```
1  #pragma once
2
3  #include "shape.h"
4  #include "shape_registry.h"
5  #include "tool_bar.h"
6
7  class draw_window final : public ml5::window
8  {
9  public: // methods
10     draw_window();
11
12 private: // methods
13     static const unsigned _MIN_OBJECT_SIZE = 5;
14
15     void on_init() override;
16
17     void on_menu(ml5::menu_event const &event) override;
18
```

```
19      void on_paint(ml5::paint_event const &event) override;
20
21      void on_mouse_left_down(ml5::mouse_event const &event) override;
22
23      void on_mouse_left_up(ml5::mouse_event const &) override;
24
25      void on_mouse_move(ml5::mouse_event const &event) override;
26
27  protected: // methods
28      void on_key(const ml5::key_event &event) override;
29
30      /**
31       * Copies the currently selected shape to the clipboard.
32       */
33      void copy_current();
34
35      /**
36       * Moves the currently selected shape from the canvas to the clipboard.
37       */
38      void cut_current();
39
40      /**
41       * Copies the clipboard contents to the canvas.
42       */
43      void paste_clipboard();
44
45  private: // methods
46      /**
47       * Prepares the canvas so that further interaction
48       * is interpreted as the intention to draw the shape
49       * as specified by the parameter.
50       * @param s The name of the drawing tool.
51       * @note This should be the key for the shape_registry (see shape_registry.h)
52       */
53      void _set_drawing_tool(const std::string &s);
54
55      /**
56       * Engages or disengages the shape selection mode
57       * according to the supplied flag.
58       * @param flag Whether the canvas should be set to
59       *             selection mode.
60       */
61      void _set_selection_mode(bool flag);
62
63  private: // members
64      /**
65       * The side bar containing selectable drawing tools.
66       */
67      std::unique_ptr <tool_bar> _tool_bar;
68
69      /**
70       * The buffer which can hold copies of drawn shapes.
71       */
72      std::unique_ptr <shape> _clipboard;
73
74      /**
75       * The currently drawn shape.
```

```
76      */
77     std::unique_ptr <shape> _shape;
78
79     /**
80      * The list of shapes that are on the canvas.
81      */
82     ml5::vector <std::unique_ptr <shape>> _shapes;
83
84     /**
85      * The currently selected shape.
86      * If no shape is selected, this
87      * is -1.
88      */
89     int _selection { -1 }; // into _shapes
90
91     /**
92      * The function used for instantiating the correct
93      * shape according to the current drawing mode.
94      */
95     shape_creator _make_shape;
96
97     /**
98      * True whenever the canvas is in selection mode.
99      */
100    bool _selecting;
101
102    /**
103     * True whenever the canvas is in selection mode and
104     * a selected shape is moved.
105     */
106    bool _moving;
107 };
```

Listing 5: draw_window.cpp

```
1  #include "draw_window.h"
2
3  #include "ellipse.h"
4  #include "rectangle.h"
5  #include "line.h"
6  #include "arrow.h"
7
8  draw_window::draw_window() : ml5::window("ML.Draw"), _tool_bar { std::
       make_unique <tool_bar>(this, 70) }
9  {
10     _make_shape = shape_registry.at("Ellipse");
11
12     _tool_bar->add_item({
13                                 "Selection",
14                                 std::make_shared <arrow>(),
15                                 [this]() { this->_set_selection_mode(true); }
16                         });
17     _tool_bar->add_item({
18                                 "Ellipse",
19                                 std::make_shared <ellipse>(),
20                                 [this]() { this->_set_drawing_tool("Ellipse")
       ; }
21                         });
```

```
22      _tool_bar->add_item({
23                                  "Line",
24                                  std::make_shared <line>(),
25                                  [this]() { this->_set_drawing_tool("Line"); }
26                          });
27      _tool_bar->add_item({
28                                  "Rectangle",
29                                  std::make_shared <rectangle>(),
30                                  [this]() { this->_set_drawing_tool("Rectangle
        "); }
31                          });
32  }
33
34  void draw_window::on_init()
35  {
36      window::on_init();
37      add_menu("&Shape", {
38              { "&Ellipse",   "draws an ellipse" },
39              { "&Line",      "draws a line" },
40              { "&Rectangle", "draws a rectangle" }
41      });
42
43      add_menu("&Edit", {
44              { "&Copy",  "Copies the selected object" },
45              { "C&ut",   "Moves the object from the canvas to the clipboard"
        },
46              { "&Paste", "Copies the object from the clipboard to the canvas."
         }
47      });
48
49      // since the selection of a tool bar item emits an
50      // event which is handled by a callback that causes a
51      // context refresh, this call has to be down here and
52      // cannot happen in the constructor.
53      _tool_bar->select("Ellipse");
54  }
55
56  void draw_window::on_menu(ml5::menu_event const &event)
57  {
58      if (event.get_title() == "Shape") {
59          _tool_bar->select(event.get_item());
60      } else if (event.get_title() == "Edit") {
61          auto &item = event.get_item();
62          if (item == "Copy" && _selection != -1) {
63              copy_current();
64          } else if (item == "Cut" && _selection != -1) {
65              cut_current();
66          } else if (item == "Paste") {
67              paste_clipboard();
68          }
69      }
70      window::on_menu(event);
71  }
72
73  void draw_window::on_paint(ml5::paint_event const &event)
74  {
75      auto &context = event.get_context();
```

```
76
77      // draw all shapes to the canvas
78      for (std::size_t i = 0; i != _shapes.size(); ++i) {
79          _shapes[i]->draw(context);
80          if (i == _selection) {
81              // selected shapes get a highlighter
82              context.SetBrush(*wxTRANSPARENT_BRUSH);
83              context.SetPen(wxPen(wxColour(100, 255, 100), 2, wxPenStyle::
        wxPENSTYLE_USER_DASH));
84              context.DrawRectangle(_shapes[i]->get_bounding_box());
85          }
86      }
87
88      // draw the "in progress" shape
89      if (_shape) {
90          _shape->draw(context);
91      }
92
93      // draw tool bar as overlay
94      _tool_bar->draw(context);
95
96      // draw keyboard shortcuts
97      context.SetTextForeground(wxColour(0x222222));
98      event.get_context().DrawText(
99              "[Ctrl+C : Copy] [Ctrl+X : Cut] [Ctrl+V : Paste]",
100             wxPoint(_tool_bar->get_bounding_box().width + 10, get_height() -
        30));
101
102     window::on_paint(event);
103 }
104
105 void draw_window::on_mouse_left_down(const ml5::mouse_event &event)
106 {
107     if (_tool_bar->get_bounding_box().Contains(event.get_position())) {
108         // delegate mouse event
109         _tool_bar->on_mouse_left_down(event);
110     } else if (_selecting) {
111         // enable dragging mechanics
112         _moving = true;
113
114         // find selected shape
115         int i = _shapes.size() - 1;
116         while (i >= 0 && !_shapes[i]->get_bounding_box().Contains(event.
        get_position())) {
117             --i;
118         }
119         _selection = i;
120         refresh();
121     } else {
122         // create a new shape
123         _shape = _make_shape(event.get_position());
124     }
125     window::on_mouse_left_down(event);
126 }
127
128 void draw_window::on_mouse_left_up(ml5::mouse_event const &event)
129 {
```

```
130     if (_shape) {
131         // add the final shape to the canvas if it is
132         // big enough to be "relevant"
133         auto box = _shape->get_bounding_box();
134         if (box.width >= _MIN_OBJECT_SIZE || box.height >= _MIN_OBJECT_SIZE)
        {
135             _shapes.add(std::move(_shape));
136         } else {
137             _shape.release();
138         }
139         refresh();
140     } else if (_selecting) {
141         _moving = false;
142     }
143     window::on_mouse_left_up(event);
144 }
145
146 void draw_window::on_mouse_move(ml5::mouse_event const &event)
147 {
148     if (_shape) {
149         // resize currently drawn shape
150         _shape->set_right_bottom(event.get_position());
151         refresh();
152     } else if (_moving && _selection >= 0) {
153         // move the selected shape
154         auto box = _shapes[_selection]->get_bounding_box();
155         box.SetTopLeft(event.get_position());
156         _shapes[_selection]->set_bounding_box(box);
157         refresh();
158     }
159     window::on_mouse_down(event);
160 }
161
162 void draw_window::_set_drawing_tool(const std::string &s)
163 {
164     _make_shape = shape_registry.at(s);
165     if (_selection != -1) {
166         // morph selected shape into another
167         // .. bit ugly but meh
168         auto box = _shapes[_selection]->get_bounding_box();
169         _shapes[_selection] = _make_shape(box.GetLeftTop());
170         _shapes[_selection]->set_right_bottom(box.GetRightBottom());
171     }
172     _set_selection_mode(false);
173     refresh();
174 }
175
176 void draw_window::on_key(const ml5::key_event &event)
177 {
178     const auto &code = event.get_key_code();
179     if (code == WXK_CONTROL_C) {
180         copy_current();
181     } else if (code == WXK_CONTROL_X) {
182         cut_current();
183     } else if (code == WXK_CONTROL_V) {
184         paste_clipboard();
185     }
```

```
186     window::on_key(event);
187 }
188
189 void draw_window::copy_current()
190 {
191     if (_selection != -1) {
192         _clipboard = _shapes[_selection]->clone();
193     }
194 }
195
196 void draw_window::cut_current()
197 {
198     if (_selection != 1) {
199         copy_current();
200         auto &ptr = _shapes[_selection];
201         _selection = -1;
202         _shapes.remove(ptr);
203         refresh();
204     }
205 }
206
207 void draw_window::paste_clipboard()
208 {
209     if (_clipboard != nullptr) {
210         auto ptr = _clipboard->clone();
211         auto box = ptr->get_bounding_box();
212         box.x += 30;
213         box.y += 30;
214         ptr->set_bounding_box(box);
215         _shapes.add(std::move(ptr));
216         // select newly added shape
217         _selection = _shapes.size() - 1;
218         refresh();
219     }
220 }
221
222 void draw_window::_set_selection_mode(bool flag)
223 {
224     this->_selecting = flag;
225     this->_selection = -1;
226     refresh();
227 }
```

Listing 6: shape.h

```
1  #pragma once
2
3  #include <ml5/ml5.h>
4
5  class shape : public ml5::object
6  {
7  public:
8      using context_t = ml5::paint_event::context_t;
9
10     shape() : _rect { 0, 0, 0, 0 }
11     {};
12
13     explicit shape(const wxPoint point) : _rect { point, point }
```

```
14      {}
15
16      explicit shape(const wxRect rect) : _rect { rect }
17      {}
18
19      /**
20       * Prepares the graphical context for drawing the shape.
21       */
22      void draw(context_t &context) const
23      {
24          context.SetPen(m_pen);
25          context.SetBrush(m_brush);
26
27          on_draw(context);
28      }
29
30      /**
31       * This method is only used for the icons within
32       * the tool bar: Selected icons are drawn "inverse".
33       */
34      virtual void draw_inverse(context_t &context) const
35      {
36          context.SetPen(m_pen);
37          context.SetBrush(*wxTRANSPARENT_BRUSH);
38          on_draw(context);
39      }
40
41      /**
42       * Adjusts width and height using the supplied corner
43       * point relative to the origin (top left).
44       */
45      void set_right_bottom(const wxPoint point)
46      {
47          _rect.SetRightBottom(point);
48      }
49
50      /**
51       * @return The rectangle surrounding the whole shape.
52       */
53      wxRect get_bounding_box() const
54      {
55          return _rect;
56      }
57
58      /**
59       * Resizes the shape so that it touches the sides of
60       * the supplied rectangle.
61       */
62      void set_bounding_box(const wxRect &rect)
63      {
64          _rect = rect;
65      }
66
67      /**
68       * @return A deep copy of the shape.
69       */
70      virtual std::unique_ptr <shape> clone() const = 0;
```

```
71
72 protected:
73
74     /**
75      * A factory method that is called when drawing preparations
76      * are finished.
77      */
78     virtual void on_draw(context_t &context) const = 0;
79
80     /**
81      * The default shape color.
82      */
83     wxBrush m_brush { wxBrush(wxColour(66, 134, 244)) };
84
85     /**
86      * The default border-color and -width of the shape.
87      */
88     wxPen m_pen { wxPen(wxColour(27, 77, 155), 2) };
89
90     /**
91      * The rectangle enclosing the shape.
92      */
93     wxRect _rect {};
94 };
```

Listing 7: ellipse.h

```
1  #pragma once
2
3  #include "shape.h"
4
5  class ellipse final : public shape
6  {
7  public: // methods
8      ellipse() = default;
9
10     explicit ellipse(const wxPoint point) : shape { point }
11     {}
12
13     explicit ellipse(const wxRect rect) : shape { rect }
14     {}
15
16     std::unique_ptr <shape> clone() const override
17     {
18         return std::make_unique <ellipse>(_rect);
19     }
20
21 private: // methods
22     void on_draw(context_t &context) const override
23     {
24         context.DrawEllipse(_rect);
25     }
26 };
```

Listing 8: line.h

```
1  #pragma once
2
```

```
3  #include "shape.h"
4
5  class line : public shape
6  {
7  public:
8      line() = default;
9
10     explicit line(const wxPoint point) : shape { point }
11     {}
12
13     explicit line(const wxRect rect) : shape { rect }
14     {}
15
16     void draw_inverse(context_t &context) const override
17     {
18         wxPen thicc { m_pen.GetColour(), m_pen.GetWidth() + 2 };
19         context.SetPen(thicc);
20         on_draw(context);
21     }
22
23     std::unique_ptr <shape> clone() const override
24     {
25         return std::make_unique <line>(_rect);
26     }
27
28 protected:
29     void on_draw(context_t &context) const override
30     {
31         context.DrawLine(_rect.GetLeftTop(), _rect.GetBottomRight());
32     }
33 };
```

Listing 9: rectangle.h

```
1  #pragma once
2
3  #include "shape.h"
4
5  class rectangle final : public shape
6  {
7  public: // methods
8      rectangle() = default;
9
10     explicit rectangle(const wxPoint point) : shape { point }
11     {}
12
13     explicit rectangle(const wxRect rect) : shape { rect }
14     {}
15
16     std::unique_ptr <shape> clone() const override
17     {
18         return std::make_unique <rectangle>(_rect);
19     }
20
21 private: // methods
22     void on_draw(context_t &context) const override
23     {
24         context.DrawRectangle(_rect);
```

```
25      }
26  };
```

Listing 10: arrow.h

```
1  #pragma once
2
3  #include "shape.h"
4
5  class arrow : public shape
6  {
7  public: // methods
8      arrow() = default;
9
10     std::unique_ptr <shape> clone() const override
11     {
12         return std::unique_ptr <arrow>();
13     }
14
15     void draw_inverse(context_t &context) const override
16     {
17         wxPen thicc { m_pen.GetColour(), m_pen.GetWidth() + 2 };
18         context.SetPen(thicc);
19         on_draw(context);
20     }
21
22 protected: // methods
23     void on_draw(context_t &context) const override
24     {
25         auto root = _rect.GetTopLeft();
26
27         // "shaft"
28         context.DrawLine(root, _rect.GetBottomRight());
29
30         // "legs"
31         auto to_right = root;
32         to_right.x += (_rect.GetWidth() / 3);
33         context.DrawLine(root, to_right);
34
35         auto to_bottom = root;
36         to_bottom.y += (_rect.GetWidth() / 3); // Using _width_ by intention!
37         context.DrawLine(root, to_bottom);
38     }
39 };
```

Listing 11: shape_registry.h

```
1  #pragma once
2
3  #include "shape.h"
4  #include "ellipse.h"
5  #include "rectangle.h"
6  #include "line.h"
7
8  /**
9   * Instantiates a concrete shape.
10  */
11 using shape_creator = std::function <std::unique_ptr <shape>(const wxPoint)>;
```

```
12
13 /**
14  * A table containing all instantiation functions, stored by the shape they create.
15  */
16 const std::map <std::string, shape_creator> shape_registry {
17         { "Ellipse",   [](const wxPoint p) { return std::make_unique <ellipse
       >(p); }},
18         { "Rectangle", [](const wxPoint p) { return std::make_unique <
       rectangle>(p); }},
19         { "Line",      [](const wxPoint p) { return std::make_unique <line>(p
       ); }}
20 };
```

Listing 12: draw_application.h

```
1 #pragma once
2
3 #include "draw_window.h"
4
5 class draw_application final : public ml5::application
6 {
7     std::unique_ptr <ml5::window> make_window() const override
8     {
9         return std::make_unique <draw_window>();
10    }
11 };
```

1.3 Tests

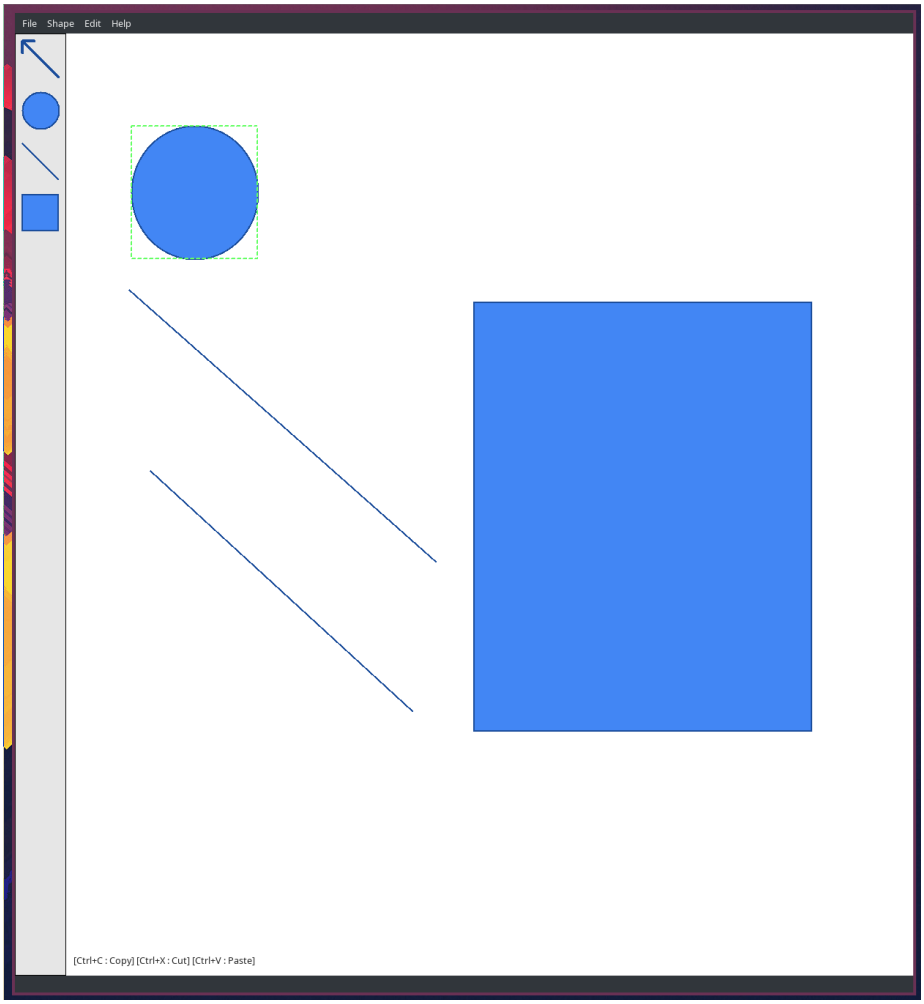


Figure 1: Alle Formen wurden gezeichnet, das Fenster ist im Selektionsmodus und dementsprechend ist der Pfeil in der *tool_bar* hervorgehoben.

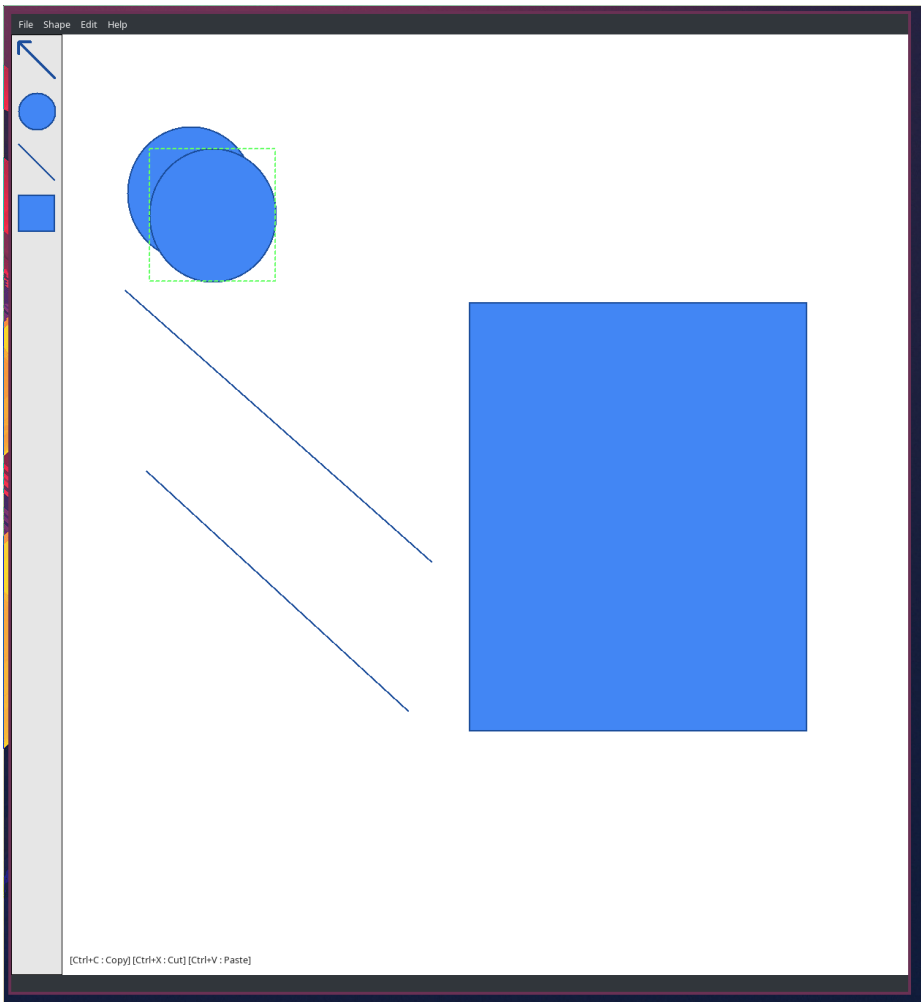


Figure 2: Die Ellipse wurde kopiert und dann (automatisch) leicht versetzt eingefügt und (ebenfalls automatisch) markiert.