# B-400-OOP_arcade

**Documentation Graphical Interface**

## Interface Game

```
class IGame {

    public:

    virtual ~IGame() = default;

    virtual void start_game() = 0;

    virtual std::string game_tick() = 0; //returns which key was pressed
this tick (or “” if non was pressed), to e.g. allow switching libs in the
core

    virtual bool is_running() = 0;

virtual void use_graphics_lib(IGraphics *lib) = 0;

virtual void reload_all_objects() = 0; //for each object saved in the game
lib (as void *) this, calls the lib->createObject function with the
corresponding creation data. Required to update all Object pointers when
loading a new Graphics lib.

protected:

IGraphics *lib; //saves the currently used graphics lib to access all it’s
functions

};
```

## Interface Graphics

```
    enum object_type {
        TEXT,
        SPRITE,
        RECT
    };

    typedef struct object_creation_data_t {
        object_type type;
        std::string text;
        std::string path_to_resource;
        std::string color_name;
    } object_creation_data;



 IGraphics {
    virtual ~IGraphics() = default;

    virtual void init_screen(int x, int y) = 0;
    virtual void destroy_screen() = 0;
    virtual void clear_screen() = 0; //clear screen so you can draw on it
again
    virtual void display_screen() = 0;
    virtual void *createObject(object_creation_data *object_data) = 0;
//create a new Text, Sprite or Rectangle, the returned pointer can be
passed to draw afterwards (you should save it in your game class!)n memory

    virtual void draw (void *object, int x0, int x1, int y0, int y1) = 0;
// draw the previously created object from point x0|y0 to point x1| y1
(see below)

    virtual void deleteObject(void *object) = 0;

    virtual std::string getPressedKey() = 0; //see below

};
```

- These Interfaces have to be included in every intermediate lib as well as in the core project!! (you are allowed to copy paste them from here)
- the libs need to implement these interface (assign all functions), member variables can be freely added while doing so

## How to load the Game/ Graphics functions

- Every Game Library need to be a child class of IGame and every Graphics Library a child class of IGraphics
    - You can access the libraries by loading this class
- this can be done by loading a function that returns an instance of the class, and using the returned class for all continuous function calls
- **The functions you have to use  (need to be in every lib, in addition to the class!)**:

```
//in the .cpp of every graphics lib:

extern "C" {

    IGraphics *create_graphics() {

        return new Graphics;

    }

}
```

```
//in the .cpp of every game lib:

extern "C" {

    IGame *create_game(IGraphics *lib) {

       return new Game(lib);

    }

}
```

- An excerpt explaining the principle behind our Class Interfaces:

  The solution is that our main program doesn't create the objects, at least not directly. The same library that provides the class derived from shape must provide a way to create objects of the **new** class. This could be done using a **factory** class, as in the factory design pattern (see Resources) or more directly using a single function. To keep things simple, we will use a single function here. The prototype for this function is the same for all shape types:

  ```
  shape *maker();
  ```

  **maker** takes no arguments and returns a pointer to the constructed object. For our hexapod class, maker might look like this:

  ```
  shape *maker(){
     return new hexapod;
  }
  ```

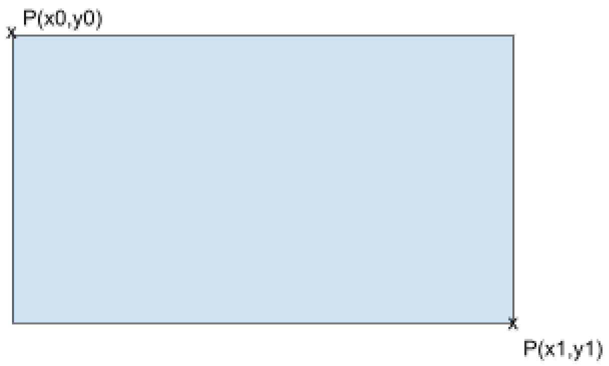
  It is perfectly legal for us to use **new** to create the object, since maker is defined in the same file as hexapod.
  Now, when we use dlopen to load a library, we can use dlsym to obtain a pointer to the maker function for that class. We can then use this pointer to construct objects of the class. For example, suppose we want to dynamically link a library called libnewshapes.so which provides the hexapod class. We proceed as follows:

  ```
  void *hndl = dlopen("libnewshapes.so", RTLD_NOW);
  if(hndl == NULL){
     cerr << dlerror() << endl;
     exit(-1);
  }
  void *mkr = dlsym(hndl, "maker");
  ```

- full article: https://www.linuxjournal.com/article/3687

## Functions contained in graphical libs

- void init_screen(x, y); // x, y in px
- void clear_screen();
- void destroy_screen();
- void *createObject(object_creation_data *object_data); // initializes object and returns pointer to it in memory
  // object_creation_data is a pointer to a type of object_creation_data
- void draw(void *object, int x0, int x1, int y0, int y1); // x, y in px



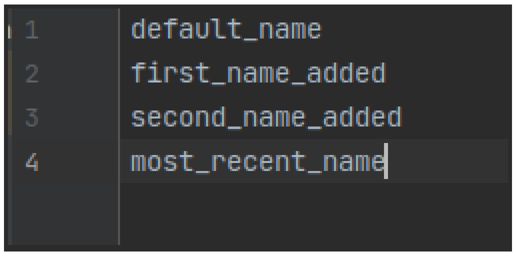
  // draws object at given parameters
- void deleteObject(void *object)
- std::string getPressedKey(void)
  // on no user input: “”
  // on window exit: “exit”
  // on pressed enter: “enter”
  // on pressed space: “space”
  // on pressed backspace: “backspace”
  // on pressed escape: “esc”
  // on pressed tabulator: “tab”
  // all keys from a-z / 0-9 are recognized
  // arrow keys are supported: “up”, “left”, “right”, “down”
  // other input/ unsupported keys: “*”
  -> e.g. on pressed ‘e’: “e”  ! only one key can be pressed at the time

## Functions contained in game libs

- void start_game()
- std::string game_tick() // returns event trigger (eg “esc”) in case core module wants to handle
- bool is_running()
- void use_graphics_lib(IGfx *lib)
- void reload_all_objects()
    - for each object saved in the game lib (as void *) this, calls the lib->createObject function with the corresponding creation data.
    - Required to update all Object pointers when loading a new Graphics lib
    - **to allow this function to work you must save the creation data of all objects created during the game within your game-lib**

## How to save/load usernames

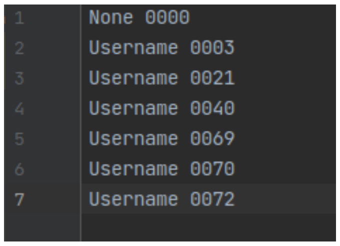
- usernames must be stored in a file called “users”
- every name must be on it’s own line, with the one last added being in the last line (without a newline afterwards)
- Example:



```
default_name
first_name_added
second_name_added
most_recent_name
```

## How to save/load high-scores

- highscores for each game must be saved in a file named "[game-name]_scores".
    - e.G. “nibbler_scores” => for a game-lib called arcade_nibbler.so
- every score must be on it’s own line, together with the the name of the user separated by a space
- highest score must be on the last line (again without a newline afterwards)
- Example:



```
None 0000
Username 0003
Username 0021
Username 0040
Username 0069
Username 0070
Username 0072
```