File - Boy.h

```h
#ifndef BOY_H
#define BOY_H
#include "../Child.h"
#include "../Parent.h"

/**
 * Represents a Boy, which is a Child with its main parent being the father.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Boy : public Child {
public:

    /**
     * Constructor for the Boy class.
     *
     * @param name The name of the child.
     * @param mother The secondary parent of the child.
     * @param father The main parent of the child.
     */
    Boy(std::string name, std::shared_ptr<Parent> mother, std::shared_ptr<Parent> father);

    /**
     * @brief Tells us if the Boy's placement in the container is valid
     * @param container to check the validity of
     * @return a Result object containing a message indicating if the Boy is alone with her mom or not.
     */
    [[nodiscard]] Result isValid(const Container& container) const override;
};

#endif  // BOY_H

```

File - Boy.cpp

```cpp
#include "Boy.h"

#include <utility>

Boy::Boy(std::string name, std::shared_ptr<Parent> mother, std::shared_ptr<Parent> father) : Child(std::move(name), std::move(father), std::move(mother)) {}

Result Boy::isValid(const Container& container) const {
    if (parentsPresenceValid(container))
        return Result::correct();

    return Result::invalid("garcon avec sa mere sans son pere");
}

```

File - Girl.h

```cpp
#ifndef GIRL_H
#define GIRL_H
#include "../Child.h"
#include "../Parent.h"

/**
 * Represents a Girl, which is a Child with its main parent being the mother.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Girl : public Child {
public:

    /**
     * Constructor for the Girl class.
     *
     * @param name The name of the child.
     * @param mother The main parent of the child.
     * @param father The secondary parent of the child.
     */
    Girl(std::string name, std::shared_ptr<Parent> mother, std::shared_ptr<Parent> father);

    /**
     * @brief Tells us if the Girl's placement in the container is valid
     * @param container to check the validity of
     * @return a Result object containing a message indicating if the Girl is alone with her dad or not.
     */
    [[nodiscard]] Result isValid(const Container& container) const override;
};

#endif  // GIRL_H

```

File - Girl.cpp

```cpp
#include "Girl.h"

#include <utility>

Girl::Girl(std::string name, std::shared_ptr<Parent> mother, std::shared_ptr<Parent> father) : Child(std::move(name), std::move(mother), std::move(father)) {}

Result Girl::isValid(const Container& container) const {
    if (parentsPresenceValid(container))
        return Result::correct();

    return Result::invalid("fille avec son pere sans sa mere");
}

```

File - Child.h

```cpp
#ifndef ENFANT_H
#define ENFANT_H
#include "../Person.h"
#include "Parent.h"

/**
 * Represents a child, which is a person with two parents.
 *
 * @Authors: Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Child : public Person {
private:
    // Pointer because we need to compare them with the container list of pointers
    std::shared_ptr<Parent> main;
    std::shared_ptr<Parent> second;
protected:

    /**
     * Checks if the main parent is present in the container. If not, it checks if the second parent is present.
     *
     * @param container The container to check for the presence of the child's parents.
     * @return true if the primary parent is present, false if the secondary parent is present without the primary one.
     */
    [[nodiscard]] bool parentsPresenceValid(const Container& container) const;

public:

    /**
     * Constructor for the Child class.
     *
     * @param name The name of the child.
     * @param main The main parent of the child.
     * @param second The secondary parent of the child.
     */
     Child(std::string name, std::shared_ptr<Parent> main, std::shared_ptr<Parent> second);
};

#endif  // ENFANT_H

```

File - Child.cpp

```cpp
#include "Child.h"

#include <utility>

Child::Child(std::string name, std::shared_ptr<Parent> main, std::shared_ptr<Parent> second) : Person(std::move(name)), main(std::move(main)), second(std::move(second)) {}

bool Child::parentsPresenceValid(const Container& container) const {
    //We assume the child is in the container since we only call this function from the Boy or Girl's "isValid()"
    if (container.getSize() <= 1)
        return true;

    // If the primary parent is here, there is no problem
    if (container.contains(main))
        return true;

    // If the primary parent is not here, this is only valid
    //  if the secondary one is not present
    return !container.contains(second);
}

```

File - Cop.h

```cpp
#ifndef COP_H
#define COP_H
#include "../Person.h"

/**
 * Represents a Cop, which is a Person that can monitor a Thief.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Cop : public Person {
public:

    /**
     * Constructor of a Cop.
     *
     * @param name The name of the Cop.
     */
    explicit Cop(std::string);

    /**
     * Tells us if the person can drive
     *
     * @return true if the person can drive, false otherwise
     */
    [[nodiscard]] bool canDrive() const override;
};

#endif  // COP_H

```

File - Cop.cpp

```cpp
#include "Cop.h"

Cop::Cop(std::string name) : Person(std::move(name)) {}

bool Cop::canDrive() const {
    return true;
}


```

File - Parent.h

```cpp
#ifndef PARENT_H
#define PARENT_H
#include "../Person.h"

/**
 * Represents a Parent, which is a Person.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Parent : public Person {
private:
public:

    /**
     * Constructor of the Parent class.
     *
     * @param name The name of the Parent.
     */
    explicit Parent(std::string);

    /**
     * Tells us if the person can drive
     *
     * @return true if the person can drive, false otherwise
     */
    [[nodiscard]] bool canDrive() const override;
};

#endif  // PARENT_H

```

File - Parent.cpp

```cpp
#include "Parent.h"

Parent::Parent(std::string name) : Person(std::move(name)) {}

bool Parent::canDrive() const {
    return true;
}


```

File - Thief.h

```cpp
#ifndef THIEF_H
#define THIEF_H
#include "../Container.h"
#include "../Person.h"
#include "Cop.h"

/**
 * Represents a Thief, which is a Person with special rules.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Thief : public Person {
private:
    std::shared_ptr<Cop> designatedCop;

public:

    /**
     * Constructor for a Thief.
     *
     * @param name The name of the Thief.
     * @param designatedCop The Cop that is designated to monitor the Thief.
     */
    Thief(std::string name, std::shared_ptr<Cop> designatedCop);

    /**
     * Checks if the thief is in the same container as the designated cop if the thief isn't alone.
     * @param container to check the validity of
     * @return a Result object containing a message indicating if the placement is valid or not
     */
    [[nodiscard]] Result isValid(const Container& container) const override;
};

#endif  // THIEF_H

```

File - Thief.cpp

```cpp
#include "Thief.h"

#include <utility>

Thief::Thief(std::string name, std::shared_ptr<Cop> designatedCop)
    : Person(std::move(name)), designatedCop(std::move(designatedCop)) {}

Result Thief::isValid(const Container& container) const {
    if (container.getSize() == 1 || container.contains(designatedCop)){
        return Result::correct();
    } else {
        return Result::invalid(getName() + " présent avec d'autres membres sans " + designatedCop->getName());
    }
}

```

File - Bank.h

```h
#ifndef BANK_H
#define BANK_H
#include "Container.h"

/**
 * Container with all the persons on the bank
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Bank : public Container {
public:

    /**
     * Constructor
     *
     * @param name the name of the bank
     */
    explicit Bank(std::string name);
};

#endif  // BANK_H

```

File - Bank.cpp

```cpp
#include "Bank.h"
#include "Container.h"

Bank::Bank(std::string name) : Container(std::move(name)) {}

```

File - Boat.h

```cpp
#ifndef BOAT_H
#define BOAT_H
#include "Bank.h"
#include "Container.h"

/**
 * Container with all the persons on the boat, it comes with
 * methods to get it's properties or move it from one bank to another.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Boat : public Container {
private:
    Bank* _current;  // Current bank on which the boat is docked
public:
    /**
     * Constructor of the boat.
     *
     * @param name Name of the boat.
     * @param current The bank on which it is docked.
     */
    Boat(std::string name, Bank* current);

    /**
     * Create a string that can represent the boat.
     *
     * @return Boat to string..
     */
    [[nodiscard]] std::string toString() const override;

    /**
     * Set the bank on which the boat is docked to a new one.
     *
     * @param current The new bank.
     */
    void setNewBank(Bank* current);

    /**
     * Getter for the current bank.
     *
     * @return The current bank.
     */
    [[nodiscard]] Bank* getCurrentBank() const;

    /**
     * Check if any of the occupants of the boat can drive it.
     *
     * @return True if any of the occupants can drive it.
     */
    [[nodiscard]] bool canMove() const;

    /**
     * Getter for the max capacity of the boat.
     *
     * @return Max capacity.
     */
    static unsigned getMaxCapacity();
};

#endif  // BOAT_H

```

File - Boat.cpp

```cpp
#include "Boat.h"
#include "Container.h"
#include "Controller.h"
#include <iostream>


Boat::Boat(std::string name, Bank* current) : Container(std::move(name)), _current(current) {}

std::string Boat::toString() const {
    return "Bateau < " + listToString() + " >";
}

void Boat::setNewBank(Bank* current) {
    this->_current = current;
}

Bank* Boat::getCurrentBank() const {
    return _current;
}

bool Boat::canMove() const {
    return std::any_of(getOccupants().begin(), getOccupants().end(), [](const std::shared_ptr<Person>& person) {
        return person->canDrive();
    });
}

unsigned int Boat::getMaxCapacity() {
    return 2;
}
```

File - Container.h

```h
#ifndef CONTAINER_H
#define CONTAINER_H
#include <iterator>
#include <list>
#include <string>
#include <vector>
#include <memory>
#include "Result.h"
class Person;

/**
 *  Container that can have occupants and methods to get its properties and check its validity.
 *
 *  @authors Junod Arthur, Roland Samuel, Häffner Edwin.
 *  @date 18.04.2024
 */
class Container {
private:
    std::string _name;
    std::list<std::shared_ptr<Person>> _occupants;

protected :
    /**
     * Getter for the list of occupants of the container.
     *
     * @return The list of occupants.
     */
    [[nodiscard]] const std::list<std::shared_ptr<Person>>& getOccupants() const;

    /**
     * Create a string to represent the list of occupants.
     *
     * @return A string representative of the occupants.
     */
    [[nodiscard]] std::string listToString() const;

public:
    /**
     * Constructor.
     *
     * @param name Name of the container.
     */
    explicit Container(std::string name);

    /**
     * Check that all of the rules of the container are respected.
     *
     * @return A Result with the validity of the container (if not valid it will give the reason through it).
     */
    [[nodiscard]] Result isValid() const;

    /**
     * Add a person to the container.
     *
     * @param p The person we want to add.
     */
    void add(const std::shared_ptr<Person>&);

    /**
     * Remove a person to the container.
     *
     * @param p The person we want to remove.
     */
    void remove(const std::shared_ptr<Person>&);

    /**
     * Find a person in the list of the occupants of the container.
     *
     * @param name Name of the person we want to find.
     * @return The person if we found it otherwise nullptr.
     */
    [[nodiscard]] std::shared_ptr<Person> findByName(const std::string& name) const;

    /**
     * Get the number of occupants of the container.
     *
     * @return The number of occupants
     */
    [[nodiscard]] std::size_t getSize() const;

```

File - Container.h

```
81      /**
82       * Check if the person given is in this container.
83       *
84       * @param searched The person we search in the container.
85       * @return true if the person was found.
86       */
87      [[nodiscard]] bool contains(std::shared_ptr<Person> searched) const;
88
89      /**
90       * Create a string to represent the container.
91       *
92       * @return A string representative of the container.
93       */
94      [[nodiscard]] virtual std::string toString() const;
95
96      /**
97       * Empty the list of occupants
98       */
99      void clear();
100 };
101
102 #include "Person.h"  //after class declaration
103
104 #endif  // CONTAINER_H
105
```

File - Container.cpp

```cpp
#include "Container.h"

#include <cstddef>
#include <algorithm>
#include "Result.h"

Container::Container(std::string name) : _name(std::move(name)) {}

Result Container::isValid() const {
    for (const auto& person : _occupants) {
        auto result = person->isValid(*this);
        if (!result.status)
            return result;
    }
    return Result::correct();
}

void Container::add(const std::shared_ptr<Person>& p) {
    _occupants.emplace_back(p);
}

void Container::remove(const std::shared_ptr<Person>& p) {
    _occupants.remove(p);
}

std::shared_ptr<Person> Container::findByName(const std::string& name) const {
    for (const auto& person : _occupants) {
        if (person->getName() == name)
            return person;
    }
    return nullptr;
}

size_t Container::getSize() const {
    return _occupants.size();
}

bool Container::contains(std::shared_ptr<Person> searched) const {
    return std::any_of(_occupants.begin(), _occupants.end(), [&](const auto& person) {
        return person == searched;
    });
}

std::string Container::toString() const {
    return  _name + ": " + listToString();
}

std::string Container::listToString() const {
    std::string result;
    int count = 0;
    for (const auto& person : _occupants) {
        if (count++ > 0)
            result += " ";
        result += person->getName();
    }
    return result;
}

const std::list<std::shared_ptr<Person>> &Container::getOccupants() const {
    return _occupants;
}

void Container::clear() {
    _occupants.clear();
}


```

File - Controller.h

```cpp
#ifndef CONTROLLER_H
#define CONTROLLER_H
#include <array>
#include <list>
#include <functional>
#include <map>
#include <memory>
#include "Boat.h"
#include "Result.h"

const int LINE_LENGTH = 58; // Length of the line we use in the display

/**
 * Represents the _controller that handles the game logic and stores the state of the game
 *
 * @Authors: Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Controller {
private:
    static const size_t LEFT = 0; // Index in the array for the left bank
    static const size_t RIGHT = 1; // Index int the array for the right bank

    std::list<std::shared_ptr<Person>> _persons; // List of all the persons available
    Boat _boat; // The container boat
    std::array<Bank, 2> _banks; // An array for the two container bank

    /**
     * Allow us to move a person from a container to another by giving the its name.
     *
     * @param name Name of the person we want to move.
     * @param from The container from which we want to find and move the person.
     * @param to The container where we want to move the person.
     * @return A Result that is correct if we could move the person (if invalid it gives the reason: didn't find the person
, rules invalid, ...).
     */
    Result movePerson(const std::string& name, Container& from, Container& to);

    /**
     * Create the error message for the person we want to find if it's not found.
     *
     * @param name The name of the person we tried to find.
     * @param from The Container from which we wanted to find the person.
     * @return A String that represent the error for the person not found.
     */
    [[nodiscard]] std::string personNotFoundMessage(const std::string& name, const Container& from) const;

    /**
     * Allow us to easily fill the left bank with all the persons.
     */
    void fillLeftBank();

    Controller();


public:

    /**
     * Allow us to get the Singleton Instance of our Controller.
     * @return The instance of Controller.
     */
    static Controller& getInstance();

    Controller(const Controller& other) = delete;
    Controller(Controller&& other) = delete;
    Controller& operator=(const Controller& other) = delete;
    Controller& operator=(Controller&& other) = delete;

    /**
     * Embark a person to the boat
     * @param name of the person to embark
     * @return Result of the operation
     */
    Result embark(const std::string& name);

    /**
     * Disembark a person from the boat
     * @param name of the person to disembark
     * @return Result of the operation
     */
```

File - Controller.h

```
80      Result disembark(const std::string& name);
81
82      /**
83       * Move the boat to the other bank
84       * @return Result of the operation
85       */
86      Result moveBoat();
87
88      /**
89       * Prints the state of the boat and the banks.
90       */
91      void print() const;
92
93      /**
94       * Reset the game by putting every person in the left bank.
95       */
96      void reset();
97
98      /**
99       * Check if the game is won.
100      * @return true if the game is won, false otherwise.
101      */
102     bool hasWon() const;
103
104 };
105
106
107 #endif  // CONTROLLER_H
108
```

File - Controller.cpp

```cpp
#include "Controller.h"

#include <iostream>
#include <array>
#include <memory>
#include "typeOfPerson/Cop.h"
#include "typeOfPerson/Thief.h"
#include "typeOfPerson/typeOfChild/Boy.h"
#include "typeOfPerson/typeOfChild/Girl.h"


Controller::Controller()
        : _banks{Bank("Gauche"), Bank("Droite")}, _boat("Bateau", nullptr) { // Init _persons of the game

    _boat.setNewBank(&_banks.at(LEFT));
    auto pere = std::make_shared<Parent>("pere");
    auto mere = std::make_shared<Parent>("mere");
    auto policier = std::make_shared<Cop>("policier");
    _persons = {pere,
                mere,
                std::make_shared<Boy>("paul", mere, pere),
                std::make_shared<Boy>("piere", mere, pere),
                std::make_shared<Girl>("julie", mere, pere),
                std::make_shared<Girl>("jeanne", mere, pere),
                policier,
                std::make_shared<Thief>("voleur", policier)};

    fillLeftBank();
}

Result Controller::movePerson(const std::string& name, Container& from, Container& to){
    const size_t FROM_INDEX = 0;
    const size_t TO_INDEX = 1;

    if (name.empty())
        return Result::invalid("Aucune personne n'a été spécifiée");
    std::shared_ptr person = from.findByName(name);
    if (!person)
        return Result::invalid(personNotFoundMessage(name,from));

    //Making a copy of from and to, so we can simulate what's happening in them
    std::array<Container, 2> simulis{Container(from), Container(to)};

    simulis.at(FROM_INDEX).remove(person);
    simulis.at(TO_INDEX).add(person);

    for(auto& simul : simulis){
        Result res = simul.isValid();
        if(!res.status){
            return res;
        }
    }

    std::swap(from, simulis.at(FROM_INDEX));
    std::swap(to, simulis.at(TO_INDEX));

    return Result::correct();
}

std::string Controller::personNotFoundMessage(const std::string& name, const Container& from) const{
    return "La personne du nom de " + name + " n'a pas été trouvée sur le " + (&from == &_boat ? "bateau" : "bord actuel");
}


Result Controller::embark(const std::string& name) {

    if(_boat.getSize() == Boat::getMaxCapacity()){
        return Result::invalid("Le bateau ne peut pas contenir plus de 2 personnes");
    }
    return movePerson(name, *_boat.getCurrentBank(), _boat);
}

Result Controller::disembark(const std::string& name) {
    return movePerson(name, _boat, *_boat.getCurrentBank());
}

Result Controller::moveBoat() {
    if(_boat.canMove()) {
        _boat.setNewBank(&_banks.at(&_banks.at(LEFT) == _boat.getCurrentBank() ? RIGHT : LEFT));
        return Result::correct();
```

File - Controller.cpp

```cpp
    } else return Result::invalid("Aucune personne embarquée ne peut conduire le bateau");
}

void Controller::print() const {
    static const std::string DELIMITER(LINE_LENGTH, '-');
    static const std::string RIVER(LINE_LENGTH, '=');

    std::cout << DELIMITER << std::endl << _banks.at(LEFT).toString() << std::endl << DELIMITER << std::endl;

    if (_boat.getCurrentBank() == &_banks.at(RIGHT)) {
        std::cout << RIVER << std::endl << _boat.toString() << std::endl;
    } else {
        std::cout << _boat.toString() << std::endl << RIVER << std::endl;
    }

    std::cout << DELIMITER << std::endl << _banks.at(RIGHT).toString() << std::endl << DELIMITER << std::endl;
}

void Controller::reset() {
    for(auto& bank : _banks)
        bank.clear();
    _boat.clear();

    fillLeftBank();
}

void Controller::fillLeftBank() {
    for (const auto& person : _persons) {
        _banks.at(LEFT).add(person);
    }
}

bool Controller::hasWon() const {  return _banks.at(RIGHT).getSize() == _persons.size();}

Controller& Controller::getInstance() {
    static Controller instance;
    return instance;
}
```

File - Person.h

```h
#ifndef PERSON_H
#define PERSON_H
#include <list>
#include <string>

#include "Container.h"
#include "Result.h"

/**
 * Represents a Person
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Person {
private:
    std::string name;

public:

    /**
     * Construct a new Person object
     *
     * @param name
     */
    explicit Person(std::string name);


    virtual ~Person() = default;

    /**
     * Tells us if the person can drive
     *
     * @return true if the person can drive, false otherwise
     */
    [[nodiscard]] virtual bool canDrive() const;

    /**
     * Tells us if the person's placement in the provided container is valid
     * @param container to check the validity of
     * @return a Result object containing a message indicating if the placement is valid or not
     */
    [[nodiscard]] virtual Result isValid(const Container& container) const;

    /**
     * Get the name of the person
     *
     * @return the name of the person
     */
    [[nodiscard]] std::string getName() const;
};

#endif  // PERSON_H

```

File - Person.cpp

```cpp
#include "Person.h"

#include <utility>

Person::Person(std::string name) : name(std::move(name)) {}

std::string Person::getName() const {
    return name;
}

bool Person::canDrive() const {
    return false;  // Default is false
}

Result Person::isValid(const Container &container) const {
    return Result::correct(); //By default is OK because no rules
}

```

File - Result.h

```cpp
#ifndef RESULT_H
#define RESULT_H

#include <optional>
#include <string>
#include <utility>

/**
 * Represents the results of operations.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class Result {
public:
    const std::optional<std::string> reason;
    const bool status;

    static Result invalid(std::string reason);
    static const Result& correct();
private:

    /**
     * Constructor of a Result object.
     * @param reason the reason of the result.
     */
    explicit Result(std::string reason);
    Result();
};

#endif  // RESULT_H

```

File - Result.cpp

```cpp
#include "Result.h"

Result::Result() : reason(), status(true) {}

Result::Result(std::string reason) : reason(std::move(reason)), status(false) {}

const Result &Result::correct() {
    static Result ok;
    return ok;
}

Result Result::invalid(std::string reason) {
    return Result(std::move(reason));
}
```

File - RiverGame.h

```cpp
#ifndef RIVER_RIVERGAME_H
#define RIVER_RIVERGAME_H


#include <functional>
#include <string>
#include <map>
#include "Controller.h"

/**
 * Contains the logic for the river game and the map of actions.
 *
 * @authors Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
class RiverGame {
private:
    int _prompt;
    bool _gameWon;
    Controller& _controller;
    bool _shouldExit; // Boolean that is put to true when we end the game.
    // The map linking the string representation of the _actions to a function that executes it.
    std::unordered_map<char, std::function<void(std::string param)>> _actions;

    /**
     * Display the error if we give it a Result that is invalid.
     *
     * @param result The Result we want to log if invalid
     */
    void logResultErrorOrPrintController(const Result& result) const;

    /**
     * Show the menu of actions
     */
    static void showMenu();
public:
    /**
     * Constructor for the RiverGame.
     */
    RiverGame();

    /**
     * Run the game logic.
     */
    void run();
};



#endif //RIVER_RIVERGAME_H

```

File - RiverGame.cpp

```cpp
//
// Created by ajun on 5/20/24.
//

#include <iostream>
#include <regex>
#include "RiverGame.h"

RiverGame::RiverGame() : _shouldExit(false), _prompt(0), _gameWon(), _controller(Controller::getInstance()){
    _actions = {
            {'p', [this](const std::string& param) { _controller.print(); }},
            {'e', [this](const std::string& param) { logResultErrorOrPrintController(_controller.embark(param)); _prompt
++;}},
            {'d', [this](const std::string& param) {
                logResultErrorOrPrintController(_controller.disembark(param));
                _gameWon = _controller.hasWon();
                _prompt++;
            }},
            {'m', [this](const std::string& param) { logResultErrorOrPrintController(_controller.moveBoat()); _prompt++;}},
            {'r', [this](const std::string& param) { _controller.reset(); _prompt = 0;}},
            {'q', [this](const std::string& param) { _shouldExit = true; }},
            {'h', [](const std::string& param) { showMenu(); }},
    };
}

void RiverGame::logResultErrorOrPrintController(const Result &result) const {
    if (!result.status && result.reason.has_value()) {
        std::cout << "### " << result.reason.value() << std::endl;
    }else{
        _controller.print();
    }
}

void RiverGame::run() {
    std::string request;
    std::regex validateWithName("^[ed] [a-zA-Z]+$"); //Makes sure we cannot have multiple space before the name
    showMenu();
    _controller.print();

    do {
        std::cout << _prompt << "> ";
        getline(std::cin, request);

        if (request.empty())
            continue;

        char action = request.at(0);
        std::string param;

        if (request.length() > 2 && std::regex_match(request, validateWithName)) {
            param = request.erase(0, 2);
        } else if (request.length() != 1) {
            std::cout << "invalid action !" << std::endl;
            continue;
        }

        auto foundAction = _actions.find(action);
        if (foundAction != _actions.cend()) {
            foundAction->second(param);
        } else {
            std::cout << "action not found !" << std::endl;
        }
    } while(!_shouldExit && !_gameWon);

    if(_gameWon){
        _controller.print();
        std::cout << "Tout le monde est du bon côté, bien joué champion !!" << std::endl;
    }
}

void RiverGame::showMenu() {
    std::cout << "p : afficher" << std::endl
              << "e <nom>: embarquer <nom>" << std::endl
              << "d <nom>: debarquer <nom>" << std::endl
              << "m : deplacer bateau" << std::endl
              << "r : reinitialiser" << std::endl
              << "q : quitter" << std::endl
              << "h : menu" << std::endl;
}
```

File - main.cpp

```cpp
#include <cstdlib>
#include "RiverGame.h"

/**
 * Main that calls the run method of the RiverGame class
 *
 * @Authors: Junod Arthur, Roland Samuel, Häffner Edwin
 * @date 18.04.2024
 */
int main() {
    RiverGame game;
    game.run();

    exit(0);
}

```