THE MAGE GAME

Design & Develepoment

THE OFFICIAL DEVELOPERS MANUAL

Written by Stefanos Stefanou

ID:27020363

The Maze Game

Design and Development

By Stefanos Stefanou
27020363

The Design
Introduction

This is the documentation for my maze game , as final assessment for the academic year 2018-2019. It is written in C++ (v14) and SFML and is a programming-skills game(refer to Scenario part for more details).It is tested under linux with SFML installed.The final product is a Clion(™) project and it has all the necessary configurations to run out-of-the-box, as well as includes a ready .out file for further inspection

Part 1:The Scenario

There are 2 kinds of interacting objects ,The ghosts(enemies) and the players(the good guys) ,
At the start of the game ,each player subscribes his own implementation of a specific class provided by the game engine itself.
In each frame a specific method will be called and this method will return the players next move based on his own algorithm.The purpose here is to reach the Treasure(Price) .
The first player who will reach the price will win the game!.

Each player starts with 3 lives. There are 2 reasons that a player can lost a life .
At first ,when the player makes an invalid move such as move outside the level ,secondly a player loses a life in case be hit by ghost.
When the player is run out of life's , ceases to exist.

Part 2:Why c++?

C++ is the language of choice for my project . This choice is based for a number of reasons but the most important one is the Object Oriented Paradigm support . A properly designed game is a complex entity , and the OOP paradigm is ideal to handle large amounts of complexity in a elegant way

Part 3 :Initial Requirements

The requirements given by the end-client(The University) was relatively few , we have the freedom to give our personal taste on our game ,so i created a list of requirements in order to create my development plan and my initial design , just like if i had a real client in front of me . The requirements was

The user requirements for the final software are...

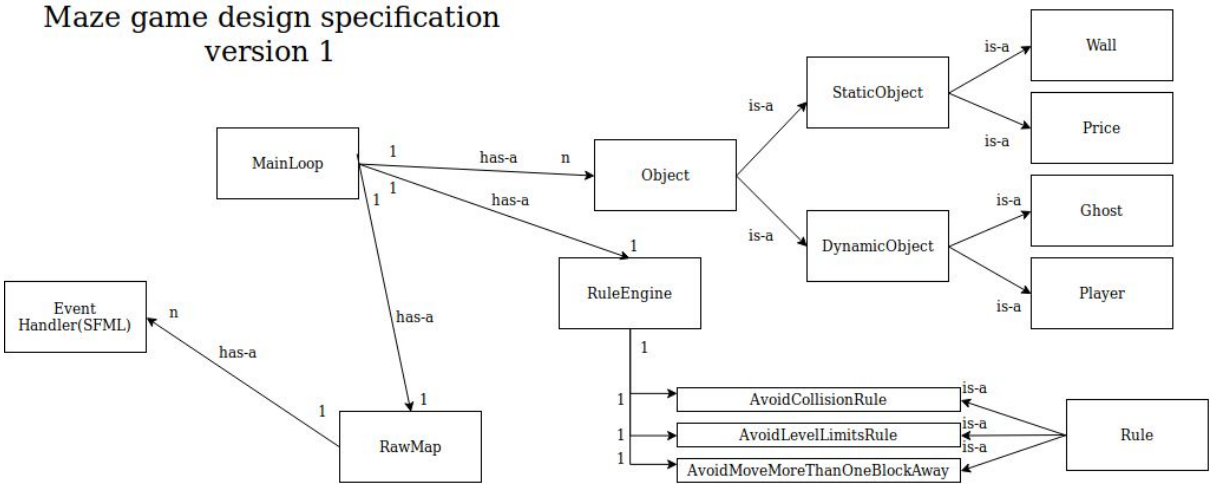
1) Be a maze game(obviously)
2) has some basic graphics(using SFML)
3) There need to be a dynamic amount of players and ghosts
4) if a ghost hits a player , the player loses a life
5) if a player hits a wall , the player loses a life
6) if a player is run out of life's , it disappears from screen
7) the maze need to be random(generated by the game engine) , but needs to be solvable every time

The System requirements for the final software are...

8) be adjusted in every screen
9) needs to be flexible and expandable
10) easy to be read and understood (source code)
11) as documented as possible(source code)
12) it needs to be easy for every programmer to write his own game class
13) it needs to have a nice and clean design

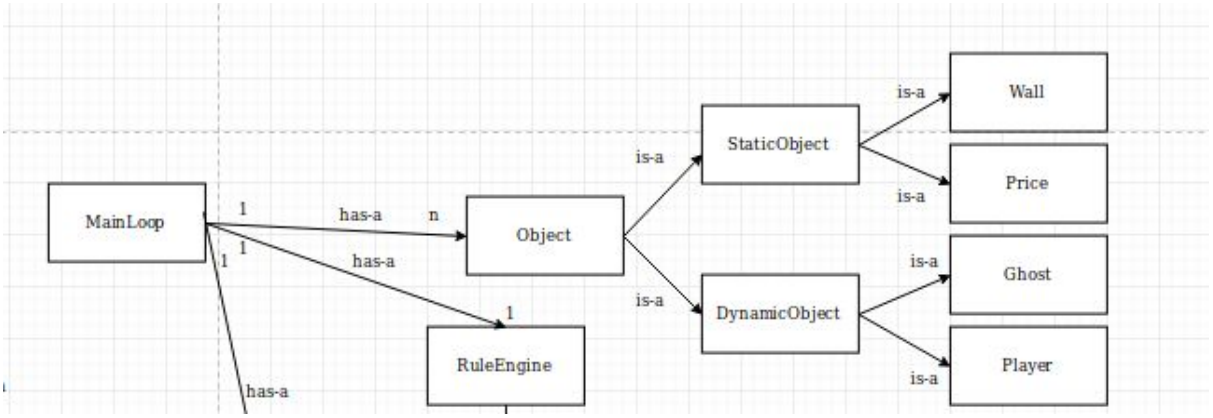
Part 4: The Basic Design

Now i will explain the way i designed my solution, emphasising in the important parts rather than the details.Here i will explain the important stuff that makes this software readable,flexible and expandable .A more analytic view of the source code exists in Part 5. The project has 4 subsystems , communicating with each other through method calling(messages) The initial design of the maze game follows



Lets start our analysis from...

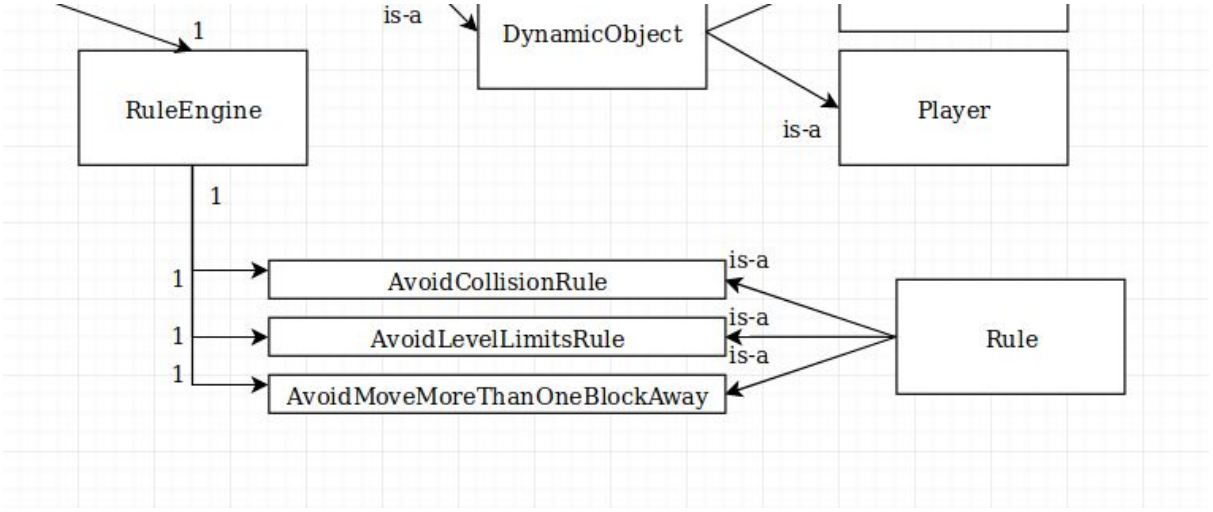
The Object class cluster



Every object on screen , is represented by a concrete implementation derived from a abstract Object class .We can see that there are two types of objects in this game , the Static Objects(Objects who don't move) and Dynamic objects(Object who can move). A number of concrete implementations is given (Wall and Price for Static objects and Ghost/Player for the dynamic ones). This cluster of classes is the barebone of the project

The move on to the first subsystem of our project …

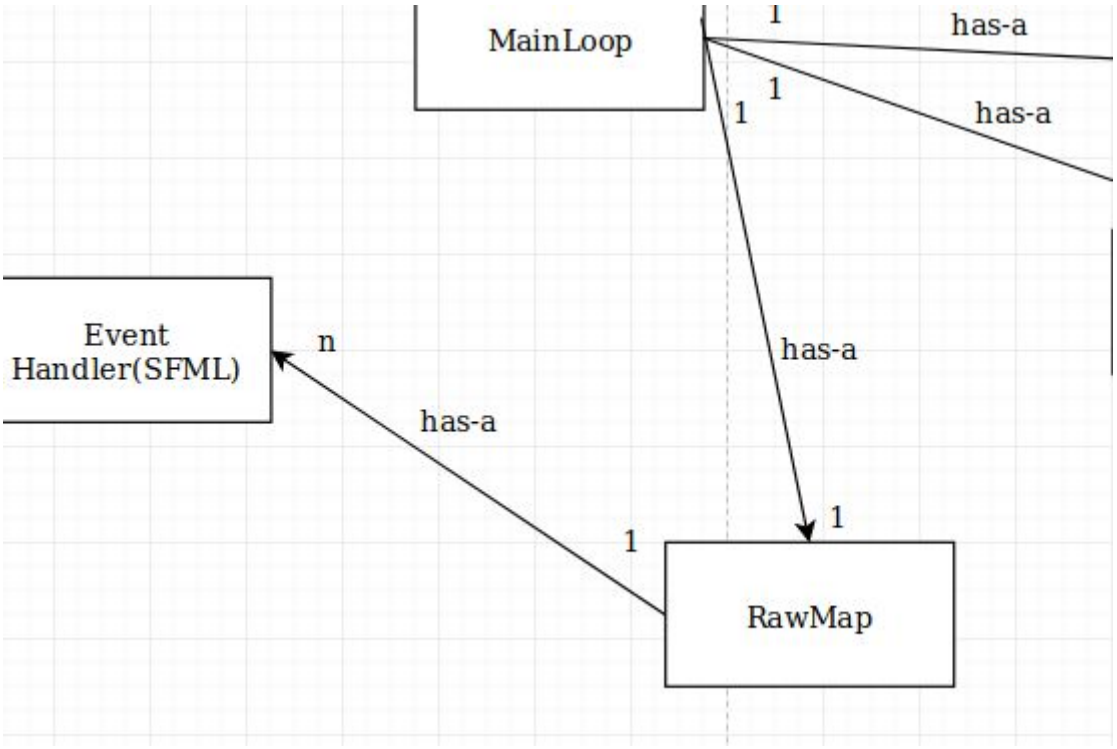
The Rule Engine



This is the *Rule Engine* . This class-cluster has the responsibility to analyse each move from each Dynamic Object and decide if violates any game rule .Any rule is a concrete implementation of the Rule Class and this check is performed right after the Game engine asks the player for a new move. If the move returned from the player does not violate any rule , then the user is moved to their new location. Please note all dynamic objects moves (Ghosts as well as players) asks the rule engine before actually any drawing on screen

Our next subsystem is…

The RawMap-Notification subsystem



The game uses a simple map system(a.k.a 2D Array) for the internal computations.The RawMap class has the responsibility to keep track of the game status and record the positions of each object in it .The RawMap system sends notifications to any interested subsystem for any event (such as “Object x moved from {x1,y1} to {x2,y2}”) .The only subsystem interested to this moment is the SFML Graphics Library ,who takes the RawMap’s events and draws to screen the result. Please note that the Event Handler runs in a different thread so there is no delay for drawing and making computations .There isn't any other connection between the main system and the SFML Graphics Library apart from this messaging system.

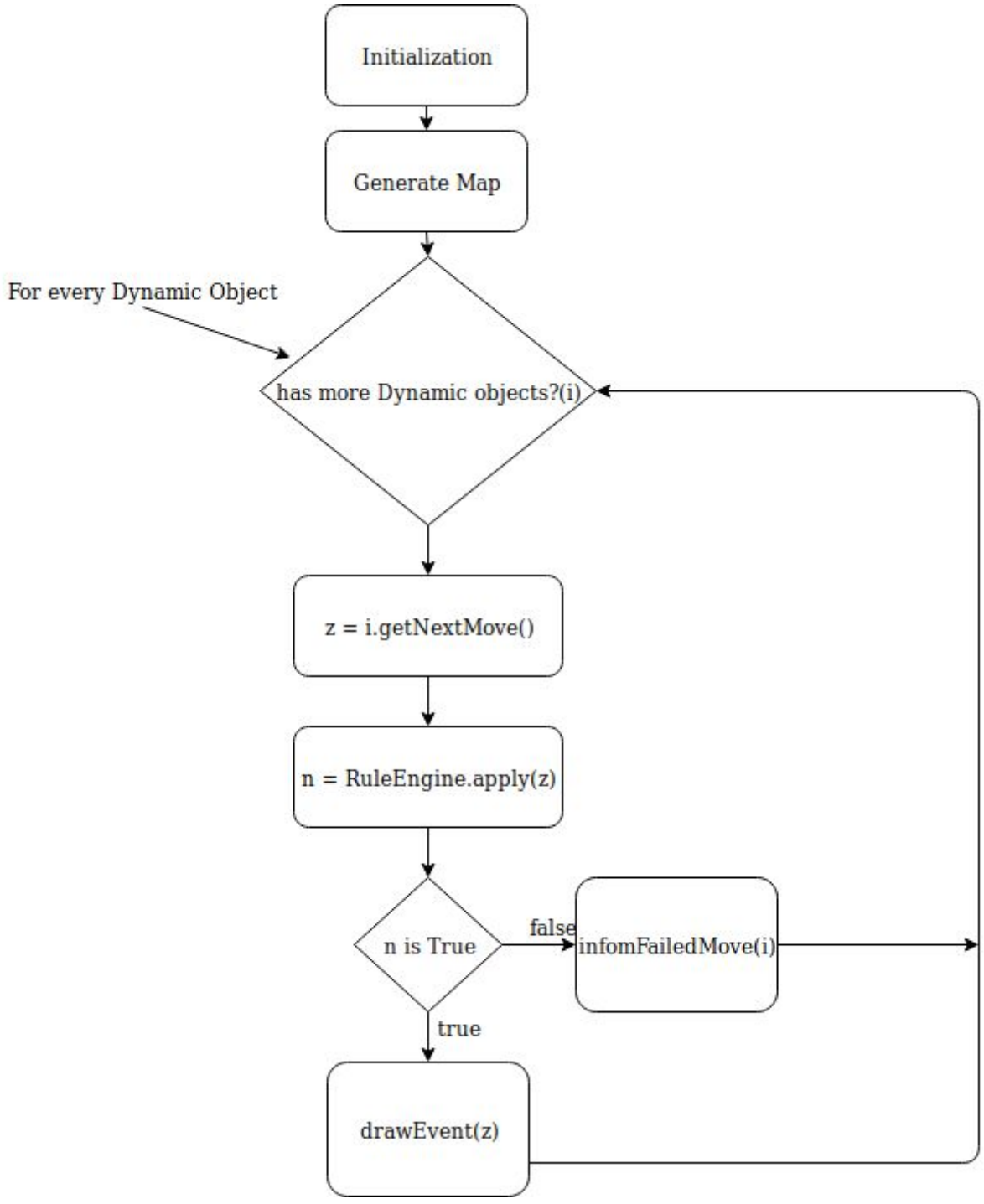
But why the need for a RawMap actually?

By handling the real computations to a seperate system ,and notify any other subsystem we can have tremendous flexibility .For example we can implement a different Event Handler who will sent the events via internet to clients ,making the system a proper server for online “maze-gaming”.We can switch graphings modules easily and improve the graphics when necessary without touch the real computation code .
Additionally by having the drawing to a separate thread, it is possible to provide steady number of frames/second , but ask the players for new moves every second.This is very important for this particular project. because we don't have to wait for user input. If the game asked the players classes for moves 30 times per second ,the game will have ended before we are able to see everything(a typical maze can be solved in under 30-40 moves)

moving towards the end of our analysis , we have

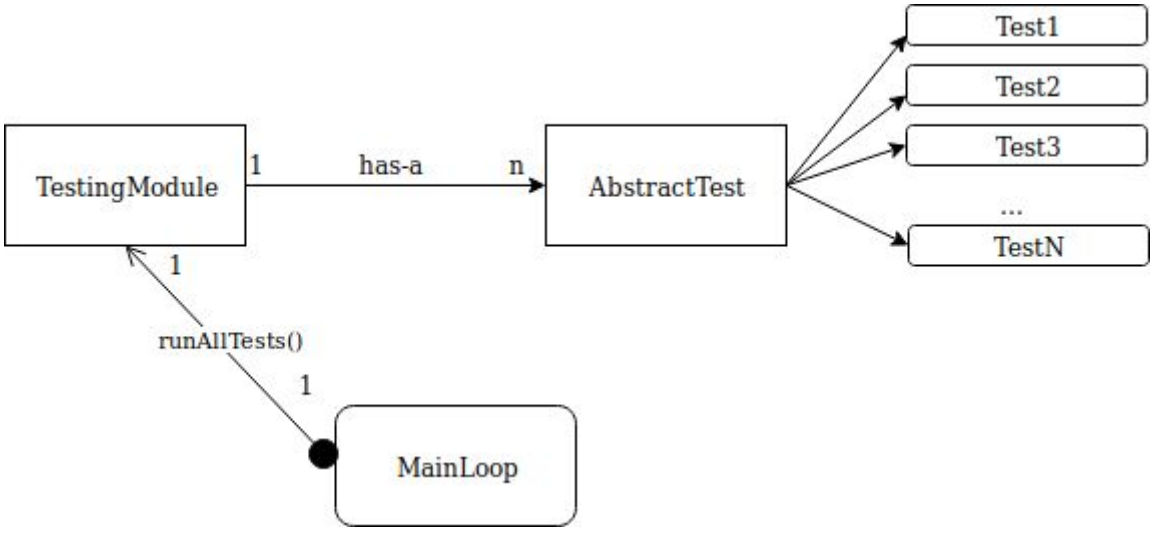
The MainLoop sybsystem

Finally ,the third subsystem we have is the MainLoop system ,this is the “glue” between every other subsystem .The only responsibility of the mainloop is to run forever(or untill all the players are dead) the following code



The Lost subsystem:Testing

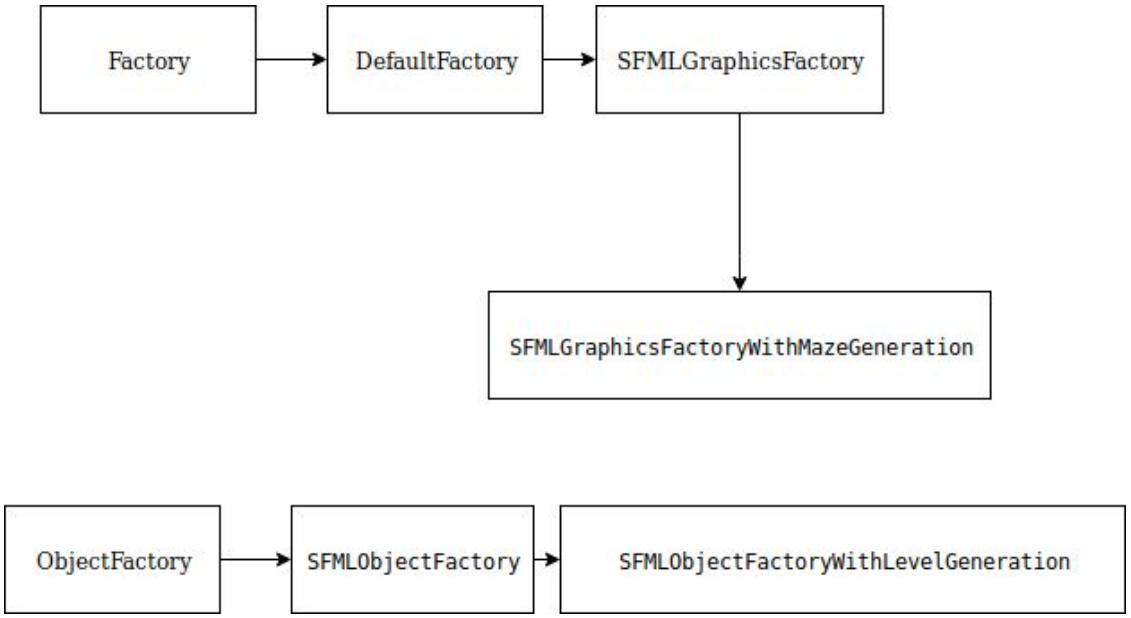
There is a fourth subsystem , the Testing Module .This subsystem has the responsibility to test all other systems parts and verify that the game can start without any problems , A simple Diagram follows



Every test is derived from the AbstractTest class . There are a number of tests(please see the Testing part below) who are executed by the TestingModule.Please note that the TestingModule runs all the tests before the MainLoop starts the game .If any of the n different tests fails , the system shuts down without proceeding

Object Creation , destruction and handling

For the enormous amount of objects this project handles , there is need for special handling in order to avoid any memory leaks and Segmentation faults. We handle all objects using factories and singleton-like methods .The factories also implement a DI Strategy (Dependency injection) by pairing up all objects dependencies.This gives us enormous flexibility because we have all objects in one place,by overriding an implementation on any of the factories , we can load a different subsystem in place of the original one , altering and testing the software with ease.Please note that there is any <<new>> statement except from those factories(there is some exceptions to this rule however),the factories have the full responsibility and ownership of every pointer in the game.The factories class cluster is given below

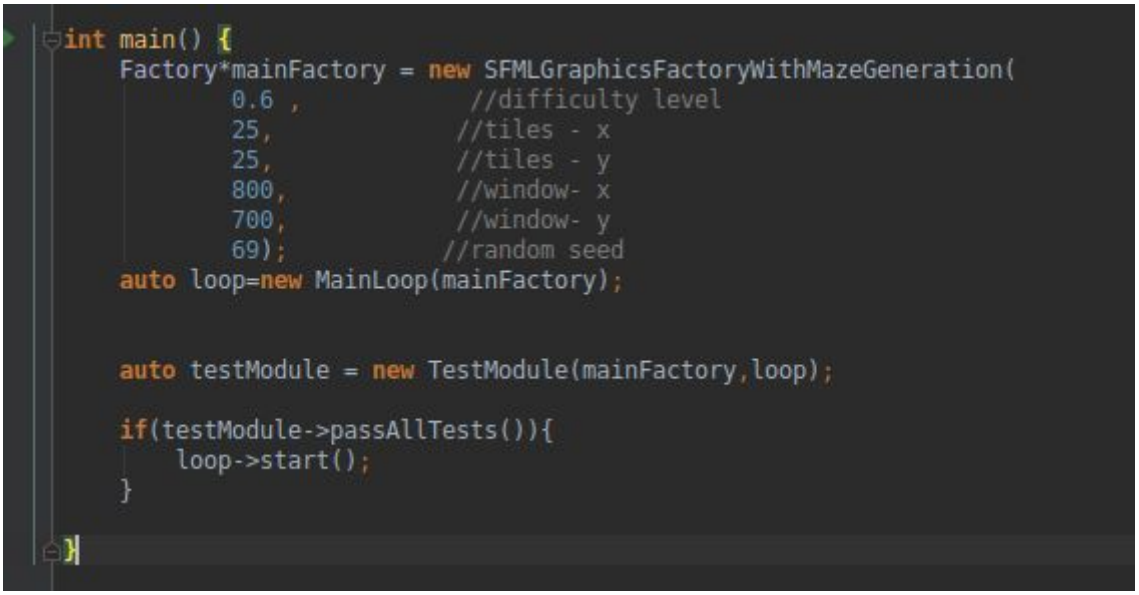


We have 2 types of factories , the System Factories(deriving from Factory Class) and the <<Object>> factories(Deriving from the ObjectFactory class) .The system factories are responsible for creating the necessary subsystems to start the game ,The Object factories are responsible for creating the interacting objects in the game(Walls,Ghosts,Players and prices!)

Let's appreciate the flexibility of this setup by thinking how it is possible to have automatic maze generation , with this set up the only thing we need to do is to override our ObjectFactory!(and Indeed , that's what SFMLObjectFactoryWithLevelGeneration do)The rest subsystem dont care on how the Wall Objects are created

Part 5: Development, a tour

Here we will analyse the development part , following the code by execution order .Lets start with our main.cpp file!



```
int main() {
    Factory*mainFactory = new SFMLGraphicsFactoryWithMazeGeneration(
            0.6 ,            //difficulty level
            25,             //tiles - x
            25,             //tiles - y
            800,            //window- x
            700,            //window- y
            69);           //random seed
    auto loop=new MainLoop(mainFactory);


    auto testModule = new TestModule(mainFactory,loop);

    if(testModule->passAllTests()){
        loop->start();
    }

}
```

A few things happening here, The first line Initializes our selected factory , This particular factory provides everything our game wants to run using the random maze generation feature. This is a nice implementation of a normal factory, taking as additional parameters a difficulty level (0.0 - 1.0) and a random seed in order to initialize the random generator(srand) . This factory creates the Wall Objects necessary to have our Random maze.

The second line creates a MainLoop object, in order to run the game properly .

After that , the Testing Module starts to evaluates all the subscribed tests, if the tests passed successfully , the game starts

```
public:
    /***
     * Checks against all tests
     * @return true on success
     */
    bool passAllTests(){
        return
            run(graphicsModuleTests())&&
            run(mainLoopTests())&&
            run(objectTests())&&
            run(rawMapTests())&&
            run(ruleEngineTests());
    }
    /***
     * Evaluates a list of tests
     * @param tests
     * @return
     */
    bool run(std::vector<AbstractTest*>tests){
        for(auto *test:tests){
            if(!test->run())return false;
            else delete(test);
        }
        return true;

    }
    /**
     * The subscribed RawMap subsystem Tests
     * @return a vector with all AbstractTests
     */
    std::vector<AbstractTest*> rawMapTests() {
        return std::vector<AbstractTest *>({
            new ApplyActionObjectTest(factory),
            new GetPointTest(factory),
            new GettersTest(factory),
            new OverridePointTest(factory)});
    }
```

As we can see , the Testing Module is very simple . There are a number of methods returning the tests for each subsystem ,a run() method to run them , and the .passAllTests() method return true if all the test passed successfully . Simplicity wins here.

Let's have a look at the Factories , before dive in the MainLoop system.

```
class Factory {
public:
    virtual ~Factory()= default;
    virtual Graphics        *getGraphicsEngine()=0;
    virtual RawMap          *getMap()=0;
    virtual RuleEngine      *getRuleEngine()=0;
    virtual ObjectFactory   *getObjectFactory()=0;

};
```

The Abstract Factory class is fairly straightforward . every method here returns the needed subsystem . As the hierarchy moves to the more specific classes the size of the factories grows ,so in order to keep the number of this document as small as possible, we will see a partial-concrete implementation , the DefaultFactory

```cpp
/*
 * The DefaultFactory class , provides the standard RawMap and RuleEngine Objects but leaves the
 * Graphics not implemented . You need to choose the more concrete implementations such as
 *  1)SFMLGraphicsFactory
 *  2)SFMLGraphicsFactoryWithMazeGeneration
 *
 */
class DefaultFactory: public Factory{
    RawMap     *map= nullptr;
    RuleEngine *ruleEngine= nullptr;
protected:
    unsigned int rawmapx,rawmapy;
public:
    DefaultFactory(unsigned int rawmapx, unsigned int rawmapy) : rawmapx(rawmapx), rawmapy(rawmapy)
    virtual ~DefaultFactory(){
        delete(map);
        map= nullptr;
        delete(ruleEngine);
        ruleEngine= nullptr;
    }
public:
    RawMap *getMap() override {
        if(map)return map;
        auto graphicsEngine=getGraphicsEngine();
        RawMap* retval= map=new RawMap(rawmapx,rawmapy);
        retval->subscribeToNotifications(graphicsEngine->getNotificationFunctor());
        return retval;
    }

    RuleEngine *getRuleEngine() override {
        return (!ruleEngine)?ruleEngine=new RuleEngine(getMap()):ruleEngine;
    }
};
```

Here we have a singleton-like factory , using a static(obviously) dependency injection pattern,as we can see at the .getMap() method.
Do you remember the Notification system that RawMap uses to acknowledge the Graphics Library? The callback functor(Function Object) is subscribed in .getMap() method.

Let's Dive in , The MainLoop code

```cpp
bool MainLoop::start() {
    return aggresiveInitialization()&&
    loadObjects()&&
    drawStaticObjects()&&
    mainLoop();
}

bool MainLoop::loadObjects() {
```

This is the entry point of MainLoop system. We can see that basically this method returns true when everything is okay , (for the reader : If you wonder why i didn't use an exception

here rather that good old bool's , i will answer , it is not worth to use a whole try{}catch statement here , we don't need additional info for the error right now , take a look at .drawDynamicObjects()). As the flow diagram says (see part 4) the only thing that mainloop does is

1. Initialization
2. Load objects to memory
3. set up the scene(.drawStaticObjects())
4. loops 'forever'(until the game is finished)

Lets dive in , and see where the 'magic' happens.

The .mainLoop() method just repeatly asks the system to draw the dynamic objects. In order to draw the dynamic objects , we need to first ask them about new moves. The class DynamicObject has a mixin class(interface) with actionCapable interface , this interface defines 2 methods , as we can see here…

```
class /*interface*/ actionCapable{
public:
    virtual Action nextActionObject(ActionObjectAdditionalInfo info)=0;
    virtual Action failedActionObjectHandler(Action ao,GenericRuleException&e)=0;
};
#endif //MAZE_IACTIONOBJECT_H
```

so in every loop , we call *->nextActionObject . Every concrete implementation returns an Action Object.

```
enum ActionType{
    //-------------------influence
    IPRICETAKEN=0,
    IPLAYERDEAD,

    //-------------------move
    MFRONT,
    MBACK,
    MRIGHT,
    MLEFT,
    MSTAND,
    MTELEPORT,

};
struct Action{
    ActionType      actionType;
    RawMapPoint     from;
    RawMapPoint     to;
    Object*  parent;
```

As we can see , the Action struct just keeps some info from the *request* of the player to move from the position {x,y} to the position {n,z} .Note that i had in mind to implement teleport capabilities , but i didn't have the necessary time :P

With this information in mind , lets see the .mainLoop() and the .drawDynamicObjects() methods of the mainLoop Subsystem

```
 *  There are 2 ways for the game to be over
 *      1)a std::exception happens (something strange , a bug maybe?)
 *      2)we are run out of players(All players are dead)
 *      3)Some player has take the price
 */
bool  MainLoop::drawDynamicObjects() {
    bool hasNoOtherPlayers=true;
    for(DynamicObject*every:*dynamicObjects){
        if (__glibc_unlikely(every->getType()==OD_PLAYER&&every->isVisible()))hasNoOtherPlayers=false;
        Action curr = every->nextActionObject(NOADDITIONALINFO);
        try{
            factory->getRuleEngine()->applyActionObject(curr);          //is the requested move a valid
            factory->getMap()->applyActionObject(curr);                 //draw it!
            every->applyQualifyingActionObject(curr);                   //inform the object for the posi

        }catch (GenericRuleException&e){
            this->error(e);                                             //logger
            Action responce=every->failedActionObjectHandler(curr,e);   //inform the object for the natu
            if(responce.actionType==IPRICETAKEN){                       //if price is taken
                return false;                                           //stop the loop
            }
        }catch (std::exception&e){
            this->error(e);
            return false;
        }

    }
    return !hasNoOtherPlayers;                                                  //loop again!
}
```

The .mainLoop() method calls every 0.3 secons the .drawDynamicObjects() method , this method does the following

1. Takes the next Action object from every DynamicObject
2. Asks the RuleEngine permission to draw the Action
3. If the RuleEngine confirms the request , forwards the request to RawMap
4. If a player hits the price , false is returned(and the .mainLoop() returns , terminating the game)
5. If there is any other player left alive, returns false

please note that the __glibc_unlikely macro is a optimisation macro for the unlikely conditions , (more here)
By continuing our tour into the games source code , we shall take a quick look at the rule engine and how it works!
The Rule engine is just a container for Rule Objects. A Rule is defined as follows

```
class Rule{

    RawMap*map;
protected:
    RawMap *getMap(){return map;}

public:
    Rule(RawMap* _map):map(_map){}
    virtual ~Rule()= default;

    virtual void ruleCheck(Action &object)=0;
    virtual std::string toString()=0;
};
```

The only interesting part here , is the .ruleCheck(Action&) method . This method is pure virtual and accepts the request of the DynamicObject for a specific move. Each concrete implementation of Rule has the chance to check his own rule by overriding this method , lets see one implementation for sake of clarity

```
class AvoidColission:    public Rule{
public:
    explicit AvoidColission(RawMap *_map) : Rule(_map){}
    virtual ~AvoidColission()= default;
    void ruleCheck(Action &object) override {
        Object*ptr;

        if((ptr=getMap()->getPoint(object.to)) &&
        ptr->isVisible())
            throw  AvoidCollisionException(object,ptr);

    }

    std::string toString() override {
        return "AvoidColission";
    }
};
```

As we can see , we check if in the requested move , there exist some other object at the moment, if this is the case , we throw a AvoidCollisionException , providing additional info to the caller(in order to understand if the object it collided with a wall or with a price :))

Returing to the Rule Engine , as a Rule Objects container , defines a single public method

```
public:
    RuleEngine(RawMap *rawMap);
    virtual ~RuleEngine()= default;

    void applyActionObject(Action&);


};
```

The .applyActionObject(Action&e) . This method check the Action against all available Rules , let's appreciate how beautiful is to load rules just like that…

```
void RuleEngine::initializeRules() {
    rules.push_back(new AvoidLevelLimits(rawMap));
    rules.push_back(new AvoidColission(rawMap));
    rules.push_back(new AvoidMoveOverOneBlockAway(rawMap));
}

void RuleEngine::applyActionObject(Action &ao) {
    for (Rule *r:rules)r->ruleCheck(ao);
}
```

Simple and elegant design

The second thing that our .drawDynamicObject() does , is that in case the move doesnt violate any rules , it draws it on screen .This is possible through the .applyActionObject of RawMap object(not to be confused with .applyActionObject of RuleEngine).
The RuleEngine::.applyActionObject(Action&) does 2 things

1. Draws to RawMap the DynamicObject
2. Informs the Graphics Module for the event

Let's have a look

```
27  void RawMap::applyActionObject(Action actionObject) {
28      overridePoint(actionObject.from, nullptr);
29      overridePoint(actionObject.to,actionObject.parent);
30      for(auto every:this->notificationCallbacks){
31          (*every)(actionObject);
32      }
33  }
```

The code here is self-explanatory . we have a std::vector of callbacks , and we call them passing as parameter the action happened.

Finally , returning to our beloved .drawDynamicObjects() , the .applyQualifyingActionObject(Action&e) informs the object for his new location

Part 6:Testing

For the purposes of the testing , my solution includes its own testing framework(in a separate 5th subsystem) . There are 13 subscribed tests who run before the start of the Game …

1. AvoidCollisionRuleTest : Any object cant be the same time in the same position with any other object

2. AvoidLevelLimitsRuleTest : Any dynamic object cant move outside the maze perimeter

3. AvoidMoveOverOneBlockAwayRuleTest : Any object cant move more than one step away of his current location with only one move instruction

4. NotificationFunctorNotNullTest : every subsystem who interested to receives notifications about RawMap events , needs to return a proper NotificationFunctor object , this Object plays the role of the “callback” function

5. AggressiveInitializationTest : The mainloop needs to sets the stage(screen initialization ,etc) properly

6. LoadObjectsTest: The Static and Dynamic Objects needs to be initialized before use

7. GetTypeTest : In any given time , it is possible to exist in the map the following objects,
   a Price , Players,Ghosts and Walls , nothing else

8. isVisibleTest: When an object is instructed to be visible , it must remain visible

9. StandStillActionObjectTest: when a object is instructed to stand still , it needs to stand still

10. ApplyActionObjectTest : When there is a valid move , it needs to be drawn to screen

11. GetPointTets : When any subsystem requests to take the object at {x,y} , the RawMap needs to return the object existing at {x,y}

12. RawMapGettersTest : The RawMap Getters needs to return the proper data

13. OverridePointTest:When any subsystem requests the {x,y} position to be overriden with the z object , after the operation the z object needs to be at {x,y}

Each Test derives from the AbstractTest class at the bottom . The TestModule is the subsystem responsible for running those tests

Lets take a taste of the tests subsystem , by looking inside the AvoidCollisionRuleTest

```
/***
 * We initialize a wall and a ghost, and we attemt to hit those
 * if the rule engine approves this move, the test faills
 */
bool applyTest() override {
    auto ruleEngine=getFactory()->getRuleEngine();
    auto map=getFactory()->getMap()->copy();
    auto wall=new Wall(RawMapPoint(0,0));
    auto *ghost=new Ghost(RawMapPoint(0,0));
    Action crashAction=DynamicObject::moveBackActionObject(ghost);

    map.overridePoint(RawMapPoint(0,0),wall);
    map.overridePoint(RawMapPoint(0,1),ghost);

    try{
        ruleEngine->applyActionObject(crashAction);
        return false;
    }catch (GenericRuleException&e){
        return true;
    }
}
```

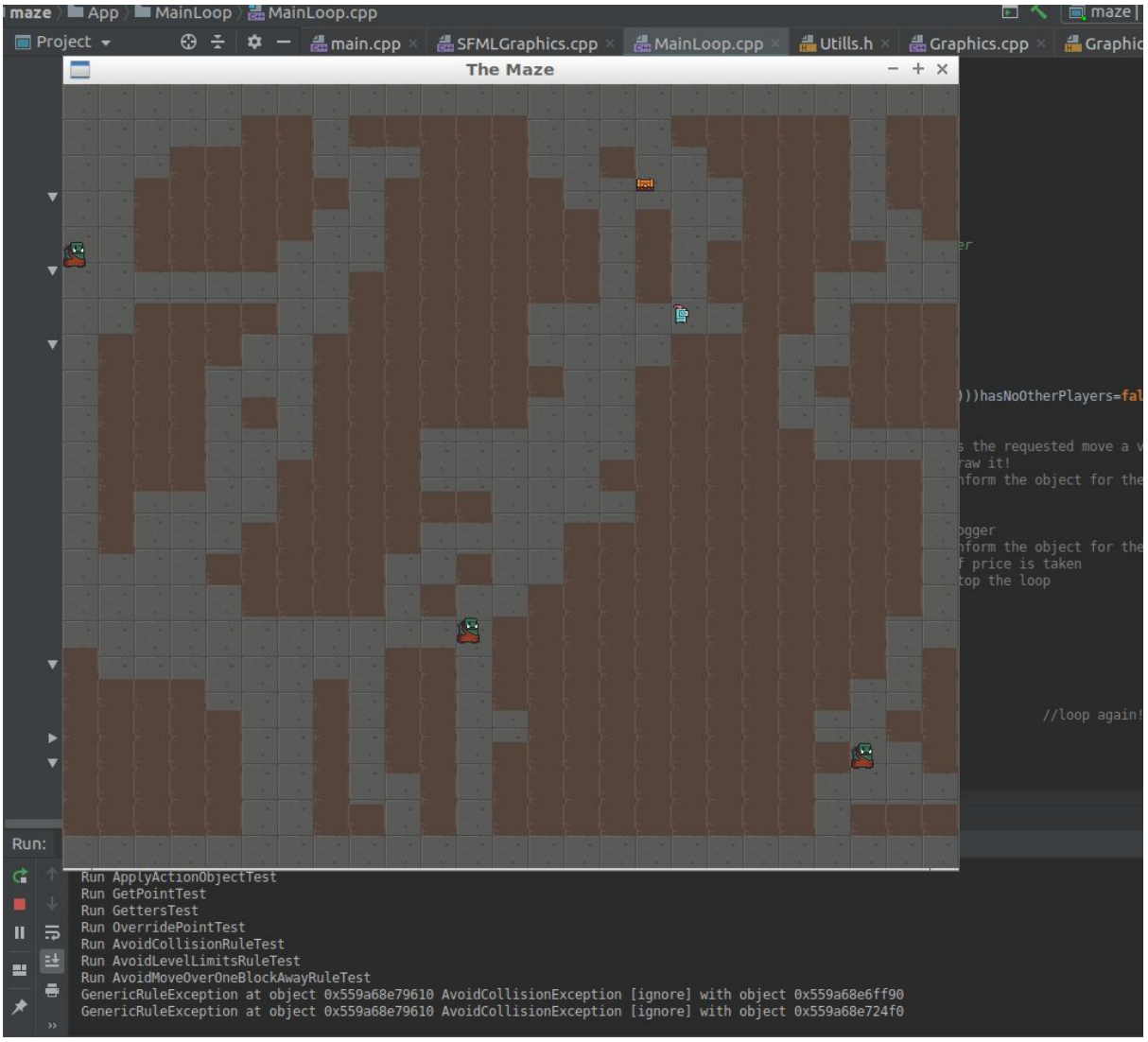
I think that the code is self-explanatory , we initially create two objects , a Wall and a Ghost , and we attempt to draw them in the same location.if the rule engine approves this move , the test fails(if it did not throw a GenericRuleException)

Part 7.1:Review

The requirements for this project make it challenging enough. I had never write a game before, neither had to play with any kind of graphics .I had to learn SFML from scratch , and to design my game to be Sprite-friendly (that's why the Action Objects and Notifications) .Additionally i wanted to have a good design because i have plans to develop it further ,after the exams .This effort lead to a nice result , but the best part is that i learned a lot. i re-study and taught myself about the GOF patterns , i learn how to properly design a OOP system , i learned how graphics works(Sprites,transformations,rotations etc) .Finally and most importanly , i learned how to attack a unknown and big problem , i learned how to methodically divide a problem into smaller , solvable parts , i learned how to learn

Part 7.2:Conclusion

It was a nice project with a lot of challenges and difficulties , not strictly by the requirements given by the assessment , but from the requirements who i choose in order to push myself in limits. i wanted a modular and flexible design ,something that it proved way harder that i first thought. I needed to re-study a ton of patterns and learn new ones on demand! it was a nice experience who gave me a lot of confidence about my programming skills , so i am happy from the result .It was the first time i had attempted to create a fully featured game .The best part was the need to apply my knowledge on linear algebra , taken from my Mathematics for computer science course , in order to transform the coordinate system from the RawMap to SFMLMap .It was the first time who i realised the role of mathematics in the computer science , Finally it was my first hands-on experience with graphics in general , and i can say that i loved it , after all , it looks good



Part8:What's included

The .zip contains

1. The Clion project files
2. An ready .out file , with one demo player loaded(just makes random moves)
3. This report booklet , who is a combined document containing Design, Development and Testing