**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**T120B516** Objektinis programų projektavimas

**Laboratorinis darbas 2**

Darbą atliko:

Ignas Lapėnas

Andrius Jankauskas

KTU 2016

# Užduotis

Realizuoti Factory (max įvertinimas 8) arba Abstract Factory (max įvertinimas 10), diagrama, pagrindimas. Panaudoti savo pasirinktą šabloną. Šablonų veikimas turi būti pristatytas per veikiantį main() metodą

# Realizacija

Suprojektuota klasių diagrama:

pav. 1 Suprojektuota klasių diagrama

# Abstract factory panaudojimas

struct Item

{

protected:

Vec2d m\_position;

public:

Item();

Item(Vec2d const& initialPosition);

virtual bool Update() = 0;

Vec2d const& GetPosition() { return m\_position; }

bool operator== (Item const& val) const { return m\_position == val.m\_position; }

};

struct ItemFactory{

virtual Item\* CreateItem() = 0;

};

class Food : public Item

{

private:

public:

Food();

bool Update();

};

struct FoodFactory : ItemFactory

{

Item\* CreateItem();

};

class Powerup : public Item {

public:

Powerup();

private:

PowerupTypes m\_type;

bool Update();

};

struct PowerupFactory : ItemFactory

{

Item\* CreateItem();

};

# Abstract factory paskirtis

Šis šablonas laidžia mums supaprastinti skirtingų objektų kvietimus runtime aplikacijoje. T.y. nereikia mums susipildyti visus įmanomus fabrikus o galime panaudoti vieną ir jį pildyti runtime metu.

# Command pattern panaudojimas

struct Enemy;

enum class MovementDirection;

typedef void(Enemy::\* enemyMethod)(MovementDirection);

struct EnemyCommand

{

private:

Enemy\* m\_enemy;

MovementDirection m\_direction;

enemyMethod m\_method;

public:

EnemyCommand(EnemyNS::Enemy\* enemy, enemyMethod method, MovementDirection direction);

void Execute();

};

enum class MovementDirection

{

none = 0,

left,

right,

up,

down

};

struct Enemy : Item

{

private:

std::queue<EnemyCommand> cmdQ;

public:

Enemy();

bool Update();

void Move(MovementDirection direction);

};

### Sukuriami priešai

void GameScene::GenerateEnemies(int difficulty)

{

for (int i = 0; i < difficulty; i++)

{

mapItems.push\_back(static\_cast<Item\*>(new EnemyNS::Enemy()));

}

}

### Command pridėjimas ir įvykdymas:

int wut = (short int)dis(gen);

cmdQ.push(EnemyCommand(this, &Enemy::Move, static\_cast<MovementDirection>(wut)));

if (cmdQ.size() > 10)

{

cmdQ.front().Execute();

cmdQ.pop();

}

# Command pattern paskirtis

Buvo panaudota, kad supaprastinti funkcijų kvietimus pagal gautus duomenis. Taip pat suteikia galimybė valdyti kitus objektus panaudojant jų funkcijas

# Strategy pattern panaudojimas

#pragma once

#include "Direction.h"

#include "Vec2d.h"

class IMoveable {

public:

virtual Vec2d Move(Vec2d headPos) = 0;

};

class Up : public IMoveable {

public:

Vec2d Move(Vec2d headPos) { return Vec2d{ headPos.x, headPos.y - 1 }; };

};

class Down : public IMoveable {

public:

Vec2d Move (Vec2d headPos) { return Vec2d{ headPos.x, headPos.y + 1 }; };

};

class Left : public IMoveable {

public:

Vec2d Move(Vec2d headPos) { return Vec2d{ headPos.x - 1, headPos.y }; };

};

class Right : public IMoveable {

public:

Vec2d Move(Vec2d headPos) { return Vec2d{ headPos.x + 1, headPos.y }; };

};

class None : public IMoveable {

public:

Vec2d Move(Vec2d headPos) { return headPos; };

};

#pragma once

#include "IMoveable.h"

#include "Vec2d.h"

#include "Direction.h"

class MoveClient

{

public:

void SetStrategy(IMoveable \*strategy);

Vec2d Move(Vec2d head) { return \_strategyClient->Move(head); }

private:

IMoveable \*\_strategyClient;

};

void MoveClient::SetStrategy(IMoveable \*strategy) {

\_strategyClient = strategy;

}

void Snake::Move()

{

m\_clearPos = m\_body.back();

MoveClient client;

Up up;

Down down;

Left left;

Right right;

None none;

switch (m\_curMoveDir)

{

case Direction::down:

client.SetStrategy(&down);

break;

case Direction::up:

client.SetStrategy(&up);

break;

case Direction::left:

client.SetStrategy(&left);

break;

case Direction::right:

client.SetStrategy(&right);

break;

case Direction::none:

client.SetStrategy(&none);

break;

}

Vec2d nextPos = client.Move(m\_body[0]);

for (int i = m\_body.size() - 1; i > 0; --i)

m\_body[i] = m\_body[i - 1];

m\_body[0] = nextPos;

}

# Strategy pattern paskirtis

Strategy pattern yra šablonas, padedantis pasirinkti algoritmą runtime metu. Mes panaudojom strategy patterna gyvatėlės judėjimui realizuoti.

# Slave pattern panaudojimas

#pragma once

class Activatable {

public:

virtual void Activate() = 0;

};

#pragma once

#include "Activatable.h"

static class ActivateServant

{

public:

ActivateServant();

~ActivateServant();

void Activate(Activatable& activate);

};

#pragma once

#include "Activatable.h"

class IncreaseSpeed :

public Activatable

{

public:

IncreaseSpeed();

~IncreaseSpeed();

void Activate();

};

#pragma once

#include "Activatable.h"

class IncreaseSize :

public Activatable

{

public:

IncreaseSize();

~IncreaseSize();

void Activate();

};

#pragma once

#include "Activatable.h"

class Reverse :

public Activatable

{

public:

Reverse();

~Reverse();

void Activate();

};

bool Powerup::Update()

{

if (m\_position == Singleton<Snake>::Instance().GetHeadPosition())

{

ActivateServant servant;

Activatable\* activate = nullptr;

IncreaseSpeed speed;

IncreaseSize size;

Reverse reverse;

switch (m\_type)

{

case PowerupTypes::Speed:

activate = &speed;

/\*for (int i = 0; i < 5; i++)

Singleton<Speed>::Instance().IncreaseLevel();\*/

break;

case PowerupTypes::Size:

activate = &size;

//Singleton<Snake>::Instance().IncreaseSize(5);

break;

case PowerupTypes::Reverse:

activate = &reverse;

//Singleton<Snake>::Instance().Reverse();

break;

default:

throw new std::exception("Something fucky was with the Powerups");

}

servant.Activate(\*activate);

return true;

}

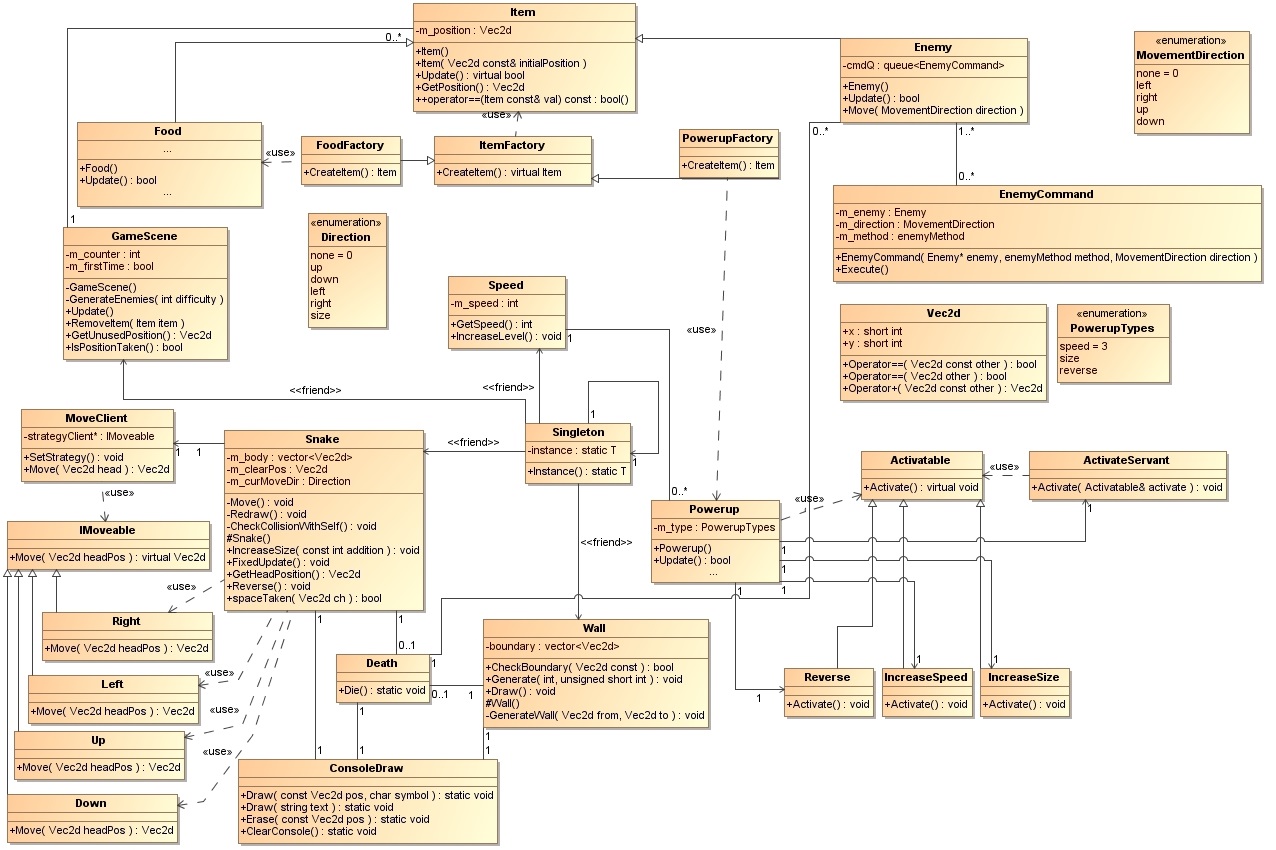
return false;

}

# Slave pattern paskirtis

Slave pattern yra struktūrinis šablonas, skirtas padėti atskirti algoritmą nuo konkrečių, panašios paskirties objektų. Algoritmai „prisegami“ runtime metu.

# Klasių diagrama



# Išvados

Skirtingi šablonai leidžia itin supaprastinti kvietimus į funkcijas gilesniuose kodo lygiuose.   
Spartesnis pildymas, naujų objektų panaudojant Abstract Factory pattern’ą.   
Command pattern leido lengvai kviesti funkcijas veliau, negu jas užregistruodavoma pasinaudojus C/C++ style pointer’iais į egzstuojančius metodus.

Strategy pattern leido velesniu metu nustatyti kaip elgsis gyvatėle, nerašant tightly-coupled code gyvatėlės viduje.

Slave pattern leido velesniu metu nurodyti kaip elgsis kokretūs powerup’ai, nerašant tightly-coupled code, leidžiant lengvai pakeisti esamų powerup’ų funkcionavimą, ar pridėti naujų powerup’ų.