KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Invaders v2

Modulio „Objektinis programų projektavimas“ laboratorinių darbų ataskaita

Darius Lapūnas, IFF-1

Kaunas, 2014

# Sistema

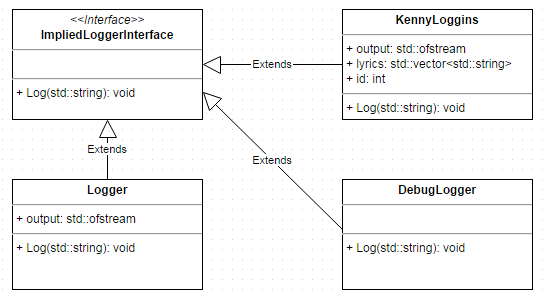
Kuriamas arkadinio stiliaus žaidimas su C++11 ir DirectX11.  
<https://github.com/DariusL/invaders2>

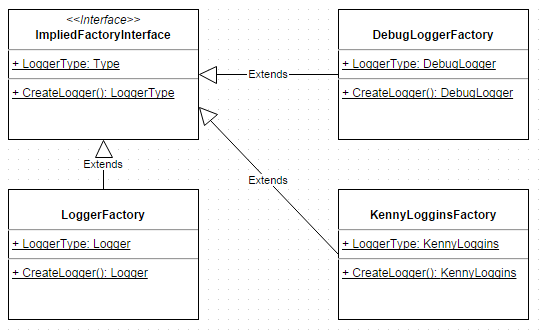
# Abstract Factory

## Šablonas

Naudojamas kai reikalinga ypač didelė objektų kūrimo abstrakcija. Kelios gamyklos implementuoja tą pačią sąsają ir kuria tos pačios sąsajos objektus. Taip galima ne tik kuriamo objekto, bet ir gamyklos tipą nustatyti vėliau.

## Diagrama





## Pagrindimas

Logger klasės turi metodą Log(std::string).LoggerFactory klasės turi statinį metodą Create() ir LoggerType tipo deklaraciją.

## Kodas

class Logger

{

std::ofstream output;

public:

Logger(std::string file);

Logger(Logger&) = delete;

Logger(Logger&&);

void Log(std::string line);

};

class DebugLogger

{

public:

DebugLogger();

void Log(std::string line);

};

class KennyLoggins

{

private:

std::vector<std::string> lyrics;

std::ofstream output;

int id;

public:

KennyLoggins();

KennyLoggins(KennyLoggins&) = delete;

KennyLoggins(KennyLoggins&&);

void Log(std::string msg);

};

class LoggerFactory

{

public:

using LoggerType = Logger;

static LoggerType Create()

{

return Logger("Log.txt");

}

};

class DebugLoggerFactory

{

public:

using LoggerType = DebugLogger;

static LoggerType Create()

{

return DebugLogger();

}

};

class KennyLogginsFactory

{

public:

using LoggerType = KennyLoggins;

static LoggerType Create()

{

return KennyLoggins();

}

};

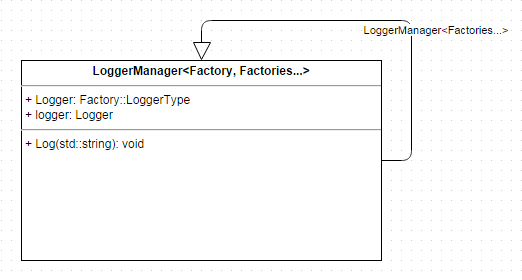
# Adapter, NullObject

## Šablonas

NullObject – objektas, kuris nieko nedaro ir reikalingas tik užimti realaus objekto vietą. Šiuo atveju naudojama užbaigti rekursiniam paveldimumui.

Adapter – sukuriamas objektas, kuris apglėbia kitą objektą. Taip įmanoma vieno tipo ar sąsajos objektus panaudoti su kitomis sąsajomis. Adapterio pagrindinė funkcija – kviesti apglėbto objekto metodus.

## diagrama



## Pagrindimas

Sukurta klasė LoggerManager, kuri sukuria žurnalus iš paduotų gamyklų. Klasė yra rekursinė ir turi po lauką kiekvienam žurnalo objektui. Tipai nustatomi iš gamyklos.

LoggerManager<> specializacija naudojama baigti rekursiją ir nieko nedaro.

## Kodas

template<typename Factory, typename... Factories>

class LoggerManager<Factory, Factories...> : public LoggerManager<Factories...>

{

friend class LoggerHelper<Factory, Factories...>;

private:

using Logger = typename Factory::LoggerType;

Logger logger;

protected:

LoggerManager()

:LoggerManager<Factories...>(),

logger(Factory::Create()){}

public:

LoggerManager(LoggerManager&) = delete;

LoggerManager(LoggerManager&&) = delete;

inline void Log(std::string msg)

{

logger.Log(msg);

LoggerManager<Factories...>::Log(msg);

}

};

template<>

class LoggerManager<>

{

friend class LoggerHelper<>;

protected:

LoggerManager(){}

public:

LoggerManager(LoggerManager&) = delete;

LoggerManager(LoggerManager&&) = delete;

inline void Log(std::string msg){}

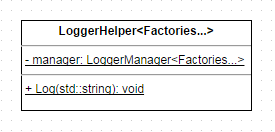
};

# Singleton

## Šablonas

Klasės objektų skaičius apribojamas iki vieno. Naudojama kaip turi būti tik vienas objektas, paprastai su kokio nors resurso alokacija. Kita priežastis – nenorima objekto visur perdavinėti ir reikia jį pasiekti statiškai. Problema – sunku valdyti jo gyvavimo laiką.

## Diagrama



## Pagrindimas

Singleton įgyvendintas C++ metodu – per statinį lokalų kintamąjį. UML to nepalaiko (kaip ir dar daug C++), todėl pažymėtas statinis laukas. Objektas sukuriamas pirmą kartą iškvietus metodą ir sunaikinamas išjungus programą. Klasė, o ne funkcija naudota todėl, kad taip galima apriboti LoggerManager konstruktorių – konstruktorius apsaugotas, o LoggerHelper pažymėta kaip friend.

## Kodas

template<typename... Factories>

class LoggerHelper

{

public:

static void Log(std::string msg)

{

static LoggerManager<Factories...> manager;

manager.Log(msg);

}

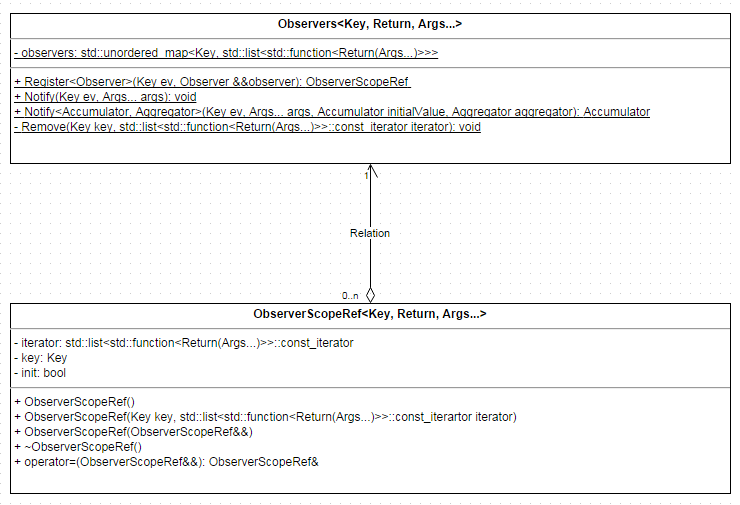
};

# Observers

## Šalbonas

Reikia objektui ar jų grupei pranešti apie būsenos pakitimus. Prie subjekto priregistruojami keli stebėtojai. Kai subjektas sužino apie būsenos pakitimą, jis visiems stebėtojams apie tai praneša – iškviečia sąsajos metodą.

## Diagrama



## Pagrindimas

Stebėtojai įgyvendinti per statinę klasę. Pats stebėtojo objektas yra std::function, arba kažkas konvertuojamo į jį. Galimas stebėtojų gražintų duomenų agregavimas su norimu akumuliatorium ir agregavimo funkcija, kurios tipas – std::function<void(Accumulator&, Return)>. Automatinis stebėtojų trynimas įgyvendintas per klasę ObserverScopeRef – sukūrus stebėtoją, gaunamas šio tipo objektas. Sunaikinus objektą sunaikinamas ir stebėtojas.

## Kodas

#pragma once

#include <list>

#include <unordered\_map>

#include <functional>

template<typename Key, typename Ret, typename... Args>

class Observers

{

public:

using Observer = std::function<Ret(Args...)>;

using ObserverList = std::list<Observer>;

using Iterator = typename ObserverList::const\_iterator;

using ObserverMap = std::unordered\_map<Key, ObserverList>;

private:

static ObserverMap observers;

public:

class ObserverScopeRef;

Observers() = delete;

static ObserverScopeRef Register(Key ev, Observer &&observer)

{

auto &f = observers[ev];

f.push\_back(std::forward<Observer>(observer));

return ObserverScopeRef(ev, std::prev(f.end()));

}

static void Notify(Key ev, Args... args)

{

auto it = observers.find(ev);

if (it != observers.end())

{

auto &obs = it->second;

for (auto &o : obs)

{

o(args...);

}

}

}

template<typename Accumulator, typename Aggregator>

static Accumulator Notify(Key ev, Args... args, Accumulator initialValue, Aggregator aggregator)

{

auto it = observers.find(ev);

if (it != observers.end())

{

auto &obs = it->second;

for (auto &o : obs)

{

aggregator(initialValue, o(args...));

}

}

return initialValue;

}

private:

static void Remove(Key key, Iterator iterator)

{

observers[key].erase(iterator);

}

public:

class ObserverScopeRef

{

private:

Iterator iterator;

Key key;

bool init;

public:

ObserverScopeRef(Key key, Iterator iterator) : key(key), iterator(iterator), init(true){}

ObserverScopeRef() : init(false){}

~ObserverScopeRef()

{

if (init)

Observers::Remove(key, iterator);

}

ObserverScopeRef(ObserverScopeRef&) = delete;

ObserverScopeRef(ObserverScopeRef &&other)

:key(other.key), iterator(other.iterator), init(other.init)

{

other.init = false;

}

ObserverScopeRef& operator=(ObserverScopeRef&) = delete;

ObserverScopeRef& operator=(ObserverScopeRef &&other)

{

if (this != &other)

{

init = other.init;

key = other.key;

iterator = other.iterator;

other.init = false;

}

return \*this;

}

};

};

template<typename Key, typename Ret, typename... Args>

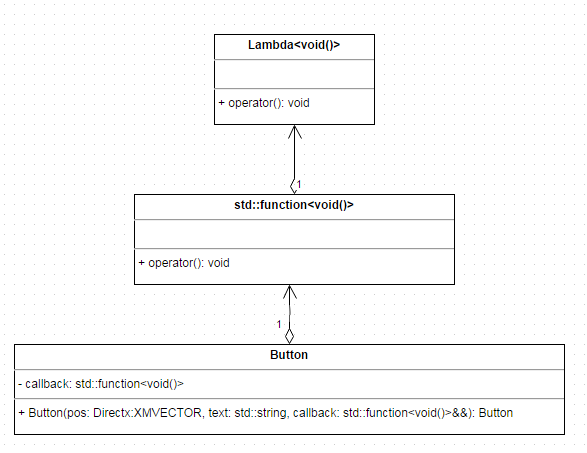
typename Observers<Key, Ret, Args...>::ObserverMap Observers<Key, Ret, Args...>::observers;

# Command

## Šablonas

Naudojamas objektas, kuris aprėpia kažkokį veiksmą. Tai paprastai yra metodas, metodo tipas ir argumentai su kuriais reikia iškviesti. Naudojamas kai reikia įvykdyti veiksmus vėlesniu laiko momentu.

## Diagrama



## Pagrindimas

Mygtukui perduodama funkcija, kurią jis turi iškviesti paspaustas – std::function. Ši klasė gali būti gauti iš std::bind, funkcijų rodyklių, lambdų. Pavyzdyje naudojamos lambdos. Taip sukuriant mygtuką nustatomas jo funkcionalumas.

## Kodas

class Button : public MenuItem, public ColorDrawableEntity

{

public:

typedef e::function<void()> PressFunction;

Button(e::XMVECTOR pos, e::string text, PressFunction &&callback);

bool Loop(InputType input);

void Render(RenderParams &params){ ColorDrawableEntity::Render(params); }

void Delay(){ clickRegister.Reset(); }

void Select(bool selected){ SetScale(selected ? 1.2f : 1.0f); }

void MoveTo(e::XMVECTOR pos){ GetEntity()->MoveTo(pos); }

private:

PressFunction callback;

PressRegister clickRegister;

};

class MainMenu : public MenuScreen

{

bool quit;

public:

MainMenu(e::XMVECTOR pos) :MenuScreen(pos, "MAIN MENU"), quit(false)

{

Add(make\_unique<Button>(XMLoadFloat3(&ZeroVec3), "START GAME", [=]{ this->child = make\_unique<GameScreen>(this->GetChildPos()); }));

Add(make\_unique<Button>(XMLoadFloat3(&ZeroVec3), "SETTINGS", [=]{ this->child = make\_unique<SettingsMenu>(this->GetChildPos()); }));

Add(make\_unique<Button>(XMLoadFloat3(&ZeroVec3), "QUIT", [=]{ this->quit = true; }));

}

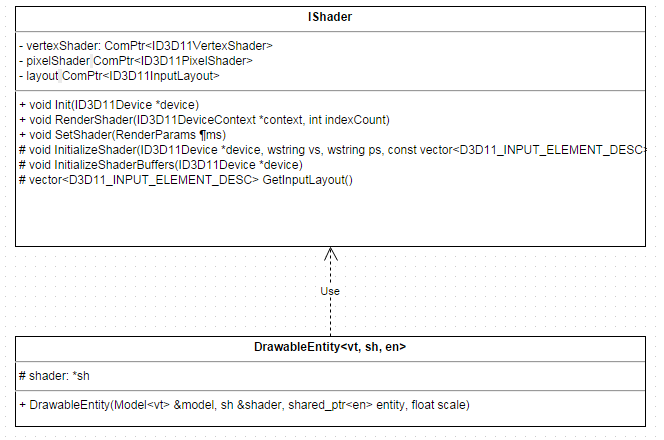
};

# Dependency Injection

## Šalbonas

Naudojamas servisas perduodamas jį naudojančiam klientui. Servisas dažnai būna kokio nors tipo resursas. Tai kartais paprasčiau implementuoti negu leisti klientui pačiam susirasti reikiamą servisą.

## Diagrama



## Pagrindimas

DrawableEntity (žaidimo objektas) perduodamas IShader, kurį naudos save piešdamas.

## Kodas

class IShader

{

public:

IShader(e::wstring vs, e::wstring ps){ this->vs = vs; this->ps = ps; }

IShader(IShader&) = delete;

IShader &operator=(IShader&) = delete;

virtual ~IShader(){}

virtual void Init(ID3D11Device \*device);

virtual void RenderShader(ID3D11DeviceContext \*context, int indexCount){context->DrawIndexed(indexCount, 0, 0);}

virtual void SetShader(RenderParams &params);

protected:

void InitializeShader(ID3D11Device \*device, e::wstring vs, e::wstring ps, const e::vector<D3D11\_INPUT\_ELEMENT\_DESC> &inputLayout);

virtual void InitializeShaderBuffers(ID3D11Device \*device) = 0;

virtual e::vector<D3D11\_INPUT\_ELEMENT\_DESC> GetInputLayout() = 0;

e::ComPtr<ID3D11VertexShader> vertexShader;

e::ComPtr<ID3D11PixelShader> pixelShader;

e::ComPtr<ID3D11InputLayout> layout;

private:

e::wstring vs, ps;

};

template<class vt, class sh, class en>

class DrawableEntity : public IDrawableObject

{

protected:

Model<vt> \*model;

e::XMFLOAT4X4 moveMatrix, scaleMatrix;

float scale;

sh \*shader;

e::shared\_ptr<en> entity;

public:

DrawableEntity(Model<vt> &model, sh &shader, e::shared\_ptr<en> entity, float scale = 1.0f);

DrawableEntity(DrawableEntity &&other);

DrawableEntity &operator=(DrawableEntity &&other);

virtual ~DrawableEntity(void);

virtual void Render(RenderParams &renderParams);

virtual void SetScale(float scale);

e::shared\_ptr<en> &GetEntity(){ return entity; }

e::XMFLOAT2 GetSize() { return model->GetSize(); }

void SetModel(Model<vt> &model){ this->model = &model; }

protected:

virtual bool Update(ID3D11DeviceContext \*context);

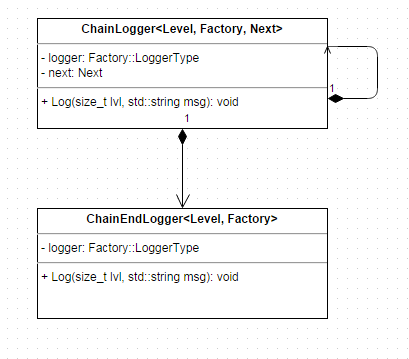
};

# Chain Of Responsibility

## Šalbonas

Objektai atliekantys kokius nors veiksmus sujungiami į grandinę. Paeiliui kviečiami grandinės metodai ir veiksmą atlieka reikiamas objektas.

## Diagrama



## Pagrindimas

ChainLogger objektas turi logger ir sekantį ChainLogger. ChainEndLogger neturi sekančio ir užbaigia grandinę.

## Kodas

template<size\_t level, typename Factory, typename Next>

class ChainLogger

{

private:

Next next;

using Logger = typename Factory::LoggerType;

Logger logger;

public:

ChainLogger()

:logger(Factory::Create()),

next(Next()){}

inline void Log(size\_t lvl, e::string msg)

{

if (lvl <= level)

logger.Log(msg);

next.Log(lvl, msg);

}

};

template<size\_t level, typename Factory>

class ChainEndLogger

{

private:

using Logger = typename Factory::LoggerType;

Logger logger;

public:

ChainEndLogger()

:logger(Factory::Create()){}

inline void Log(size\_t lvl, e::string msg)

{

if (lvl <= level)

logger.Log(msg);

}

};