KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Invaders v2

Modulio „Objektinis programų projektavimas“ laboratorinių darbų ataskaita

Darius Lapūnas, IFF-1

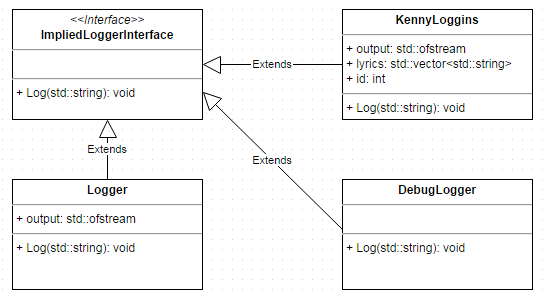
Kaunas, 2014

# Sistema

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

# Adapter

## Diagrama



## Pagrindimas

Skirtingi registravimo metodai gali būti naudojami, todėl registravimo klasė pasiekiama per sąsają. Klasės naudojamos per šablonus, todėl sąsaja nėra formaliai aprašyta. Kompiliatorius užtikrina Log metodo būvimą objektuose.

## 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;

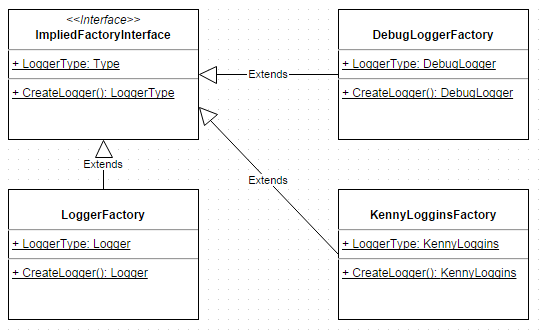
KennyLoggins(KennyLoggins&&);

void Log(std::string msg);

};

# Abstract Factory

## Diagrama



## Pagrindimas

Sąsają sudaro dvi dalys: Create() metodas ir LoggerType tipo deklaracija. Tai leidžia panaudojant per šabloną sukurti reikiamą lauką žurnalo objektui. Taip pat šablonai leidžia įgyvendinti abstrakčią gamyklą per statinį metodą.

## Kodas

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()

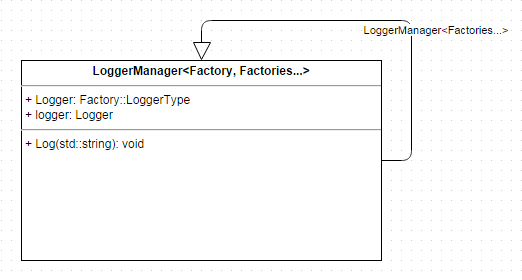
{

return KennyLoggins();

}

};

# Apjungimas



## Pagrindimas

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

## 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;

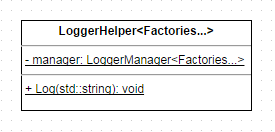
LoggerManager(LoggerManager&&) = delete;

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

};

# Singleton

## 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;

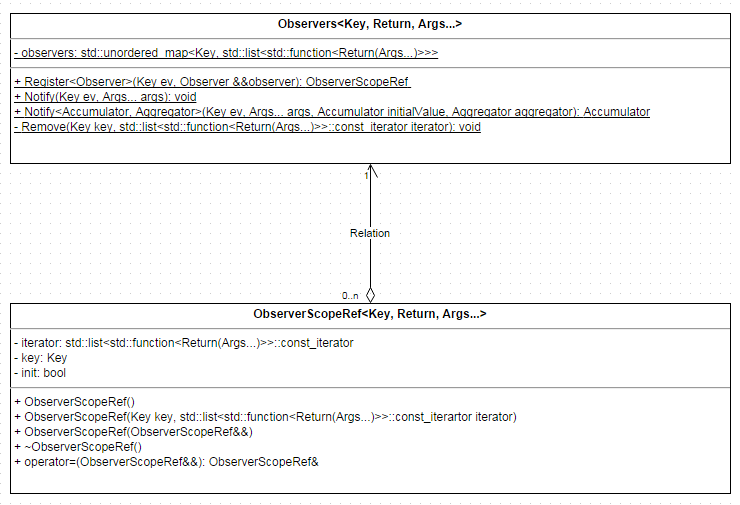
manager.Log(msg);

}

};

# Observers

## 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;