МИНОБРНАУКИ РОССИИ

Федеральное государственное образовательное

учреждение высшего профессионального образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра Математического обеспечения и применения ЭВМ

**ОТЧЕТ**

**по лабораторной работе №1**

на тему «Исследование паттернов проектирования, систем контроля версий и систем сборки проектов на примере Git и CMake»

по курсу «Архитектура и проектирование программных систем»

Выполнил:

студент группы КТбо4-12:

Паньшин А.А.

Проверил:

ассистент каф. МОП ЭВМ

Данилов Игорь Геннадьевич

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2016г.

Таганрог 2016

Содержание

[Цель работы 3](#_Toc441510741)

[Задание 3](#_Toc441510742)

[1 Описание паттернов 3](#_Toc441510743)

[1.1 Паттерн Facade 3](#_Toc441510744)

[1.2 Паттерн State 5](#_Toc441510745)

[2 Структура проекта 8](#_Toc441510746)

[3.1 Файлы CMake 11](#_Toc441510747)

[Листинг программы 12](#_Toc441510748)

# Цель работы

Исследование паттернов проектирования, систем контроля версий и систем сборки проектов на примере Git и CMake.

# Задание

3. Паттерн Faсade (фасад) и паттерн State (состояние).

# Описание паттернов

## Паттерн Facade

* + 1. **Классификация**

Фасад относится к структурным паттернам.

* + 1. **Назначение**

Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.

Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом.

* + 1. **Мотивация**

Паттерн Facade инкапсулирует сложную подсистему в единственный интерфейсный объект. Это позволяет сократить время изучения подсистемы, а также способствует уменьшению степени связанности между подсистемой и потенциально большим количеством клиентов. С другой стороны, если фасад является единственной точкой доступа к подсистеме, то он будет ограничивать возможности, которые могут понадобиться "продвинутым" пользователям.

Объект Facade, реализующий функции посредника, должен оставаться довольно простым и не быть всезнающим "оракулом".

* + 1. **Участники**

Facade (Cars) – фасад.

* Осведомлен о том, каким классам подсистемы адресовать запрос.
* Делегирует запросы клиентов подходящим объектам внутри подсистемы. codelab.ru очник

Классы подсистемы (ICar, IState, BMW, VAZ2115 и т.д.).

* Реализуют функциональность подсистемы.
* Выполняют работу, запрошенную объектом Facade, которую в свою очередь запросил у Facade один из клиентов.
* Ничего не «знают» о существовании самого фасада, то есть не хранят ссылок на него.
  + 1. **Отношения**

Клиенты общаются с подсистемой через Facade. При получении запроса от клиента объект Facade переадресует его нужному компоненту подсистемы. Для клиентов компоненты подсистемы остаются "тайной, покрытой мраком".

* + 1. **Результаты**

Преимущества Facade: codelab.ru источник оригинал codelab.ru

* Изолирует клиентов от компонентов подсистемыodelab.ru codelab.ru оригинал источник
* Позволяет ослабить связанность между подсистемой и ее клиентами.  
  Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. источник codelab.ru оригинал codelab.ru
* Фасад не исключает возможности приложениям напрямую обращаться к классам подсистемы, если это необходимо.
  + 1. **Известные применения**

Фасад удобно использовать в следующих случаях:

* Необходимо предоставить простой интерфейс к сложной подсистеме.
* Между клиентами и классами реализации абстракции существует много зависимостей.
* Необходимо расслоить подсистему (упростить).

В каркасе ЕТ++ приложение может иметь встроенные средства инспектирования объектов во время выполнения. Они реализуются в отдельной подсистеме, включающей класс фасада с именем **ProgrammingEnvironment**. Этот фасад определяет такие операции, как **InspectObject** и **InspectClass**для доступа к инспекторам.

Приложение, написанное в среде ЕТ++**,** может также запретить поддержку инспектирования. В таком случае класс **ProgrammingEnvironment** реализует соответствующие запросы как пустые операции, не делающие ничего. Только подкласс **ETProgrammingEnvironment** реализует эти операции так, что они отображают окна соответствующих инспекторов. Приложению неизвестно, доступно инспектирование или нет. Здесь мы встречаем пример абстрактной связанности между приложением и подсистемой инспектирования.

В операционной системе Choices (рис. 1) фасады используются для составления одного каркаса из нескольких. Ключевыми абстракциями в системе Choices являются процессы, память и адресные пространства. Для каждой из них есть соответствующая подсистема, реализованная в виде каркаса. Это обеспечивает поддержку переноса Choices на разные аппаратные платформы. У двух таких подсистем есть «представители», то есть фасады. Они называются **FileSystemlnterface** (память) и **Domain** (адресные пространства).

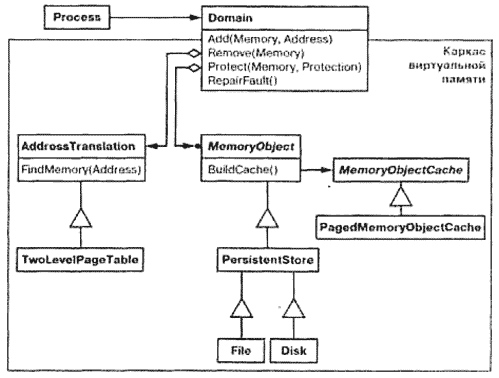


Рисунок 1 Структура каркаса ET++

Например, для каркаса виртуальной памяти фасадом служит **Domain**. Класс Domain представляет адресное пространство. Он обеспечивает отображение между виртуальными адресами и смещениями объектов в памяти, файле или на устройстве длительного хранения. Базовые операции класса Domain поддерживают добавление объекта в память по указанному адресу, удаление объекта из памяти и обработку ошибок отсутствия страниц

* Как видно из вышеприведенной диаграммы, внутри подсистемы виртуальной оригинал памяти используются следующие компоненты:
* **MemoryObject** представляет объекты данных;
* **MemoryObjectCache** кэширует данные из объектов **MemoryObj** ects в физической памяти.
* **MemoryObjectCache** - это не что иное, как объект Стратегия, в котором локализована политика кэширования;
* **AddressTranslation** инкапсулирует особенности оборудования трансляции адресов.

Операция **RepairFault** вызывается при возникновении ошибки из-за отсутствия страницы. **Domain** находит объект в памяти по адресу, где произошла ошибка и делегирует операцию **RepairFault** кэшу, ассоциированному с этим объектом. Поведение объектов **Domain** можно настроить, заменив их компоненты.

* + 1. **Родственные паттерны**

Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Помните, Adapter делает работающими вместе два существующих интерфейса, не создавая новых.

Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как сделать один объект, представляющий целую подсистему.

Mediator похож на Facade тем, что абстрагирует функциональность существующих классов. Однако Mediator централизует функциональность между объектами-коллегами, не присущую ни одному из них. Коллеги обмениваются информацией друг с другом через Mediator. С другой стороны, Facade определяет простой интерфейс к подсистеме, не добавляет новой функциональности и не известен классам подсистемы.

Abstract Factory может применяться как альтернатива Facade для сокрытия платформенно-зависимых классов.

Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.

Adapter и Facade в являются "обертками", однако эти "обертки" разных типов. Цель Facade – создание более простого интерфейса, цель Adapter – адаптация существующего интерфейса. Facade обычно "обертывает" несколько объектов, Adapter "обертывает" один объект.

## Паттерн State

* + 1. **Классификация**

Состояние - паттерн поведения объектов, задающий разную функциональность в зависимости от внутреннего состояния объекта.

* + 1. **Назначение**

Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Создается впечатление, что объект изменил свой класс.

Паттерн State является объектно-ориентированной реализацией конечного автомата.

* + 1. **Мотивация**

Основная идея паттерна – в создании абстрактного класса ICar для хранения состояний автомобиля. Данный класс является интерфейсом, единым для всех классов с различными состояниями.

* + 1. **Участники**

Context (ICar) – контекст.

* Определяет единый интерфейс для клиентов.
* Хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние.

State (IState) – состояние.

* Определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context.

Подклассы ConcreteState (MotorWorksState, MotorStoppedState) - конкретное состояние.

* Каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста Context.
* Все подклассы реализованы как Singleton с классом Destoyer, что позволяет значительно упростить их создание и работу с ними, а также инициализацию и разрушение.
  + 1. **Отношения**

Отношения между классами представлены на схеме 1:

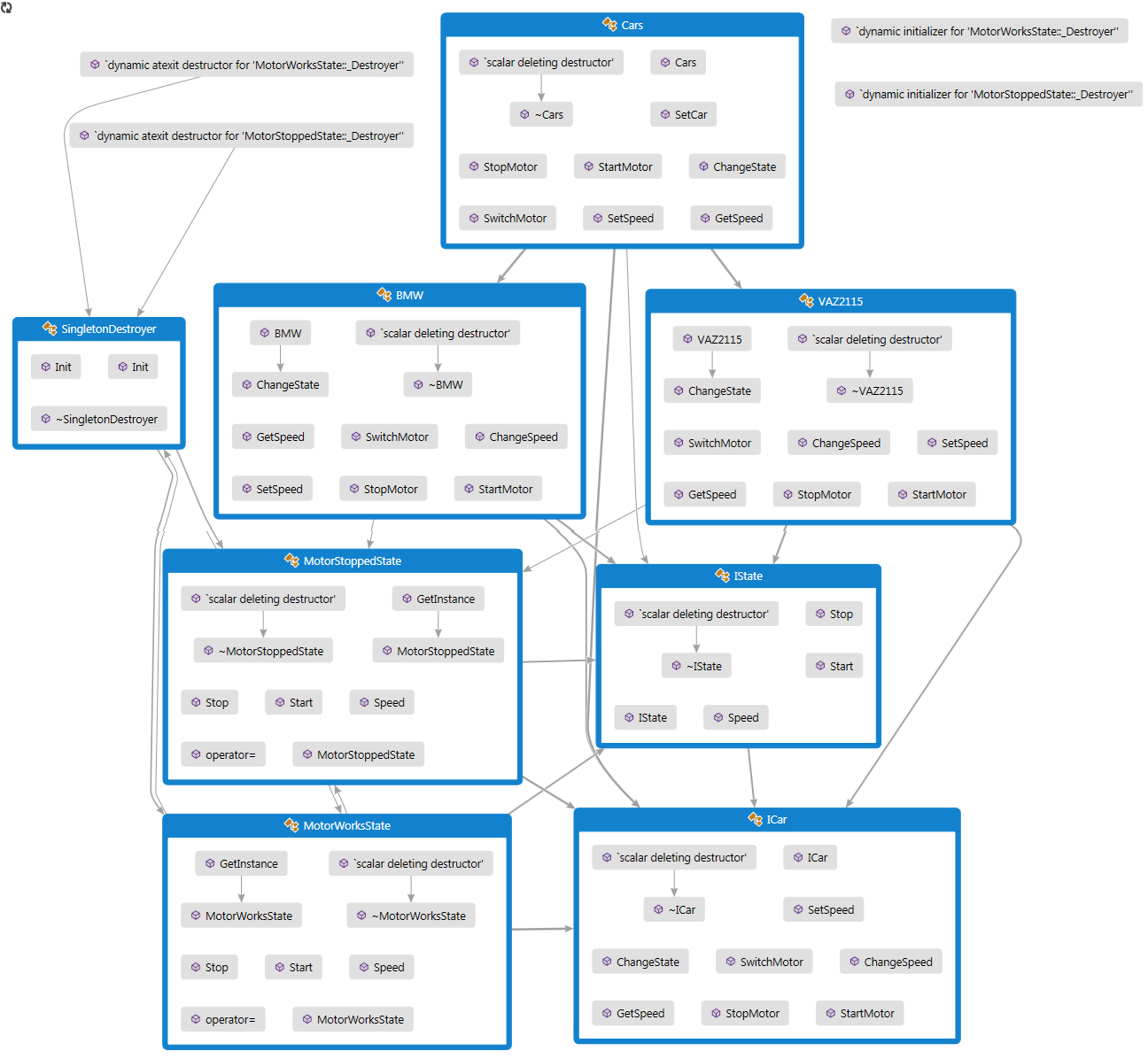


Схема 1 Отношения между классами

* + 1. **Результаты**

Паттерн State в результате:

* Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям. State помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса **State**, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов.

State увеличивает число классов. Конечно, один класс компактнее, но если состояний много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами.

Наличие громоздких условных операторов нежелательно, равно как и наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. State предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы **if** или **switch**, а распределена между подклассами **State**. При инкапсуляции каждого перехода и действия в класс – состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение.

* Делает явными переходы между состояниями.

Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты **State** могут защитить контекст **Context** от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста – это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной **State** в классе **Context**), а не нескольких.

* Объекты состояния можно разделять.

Если в объекте состояния **State** отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект **State**. Когда состояния разделяются таким образом, они являются, по сути дела, приспособленцами, у которых нет внутреннего состояния, а есть только поведение.

* + 1. **Известные применения**

Ральф Джонсон и Джонатан Цвейг характеризуют паттерн состояние и описывают его применительно к протоколу TCP. Наиболее популярные интерактивные программы рисования предоставляют «инструменты» для выполнения операций прямым манипулированием. Например, инструмент для рисования линий позволяет пользователю щелкнуть в произвольной точке мышью, а затем, перемещая мышь, провести из этой точки линию. Инструмент для выбора позволяет выбирать некоторые фигуры. Обычно все имеющиеся инструменты размещаются в палитре. Работа пользователя заключается в том, чтобы выбрать и применить инструмент, но на самом деле поведение редактора варьируется при смене инструмента: посредством инструмента для рисования мы создаем фигуры, при помощи инструмента выбора - выбираем их и т.д. Чтобы отразить зависимость поведения редактора от текущего инструмента, можно воспользоваться паттерном State.

Можно определить абстрактный класс **Tool**, подклассы которого реализуют зависящее от инструмента поведение. Графический редактор, представленный на рисунке 2, хранит ссылку на текущий объект **Tool** и делегирует ему поступающие запросы. При выборе инструмента редактор использует другой объект, что приводит к изменению поведения.

Данная техника используется в каркасах графических редакторов HotDraw и Unidraw. Она позволяет клиентам легко определять новые виды инструментов. В **HotDraw** класс **DrawingController** переадресует запросы текущему объекту **Tool**. В **Unidraw** соответствующие классы называются**Viewer** и **Tool**. На приведенной ниже диаграмме классов схематично представлены интерфейсы классов **Tool** и **DrawingController**:

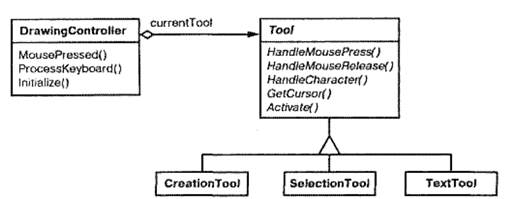


Рисунок 2 Паттерн State в графическом редакторе

Описанный Джеймсом Коплиеном прием конверт-письмо (EnvelopeLetter) также относится к паттерну State. Техника конверт-письмо – это способ изменить класс объекта во время выполнения. Паттерн State является частным случаем, в нем акцент делается на работу с объектами, поведение которых зависит от состояния.

* + 1. **Родственные паттерны**

Объекты класса State часто бывают одиночками.

Flyweight показывает, как и когда можно разделять объекты State.

Паттерн Interpreter может использовать State для определения контекстов при синтаксическом разборе.

Паттерны State и Bridge имеют схожие структуры за исключением того, что Bridge допускает иерархию классов-конвертов (аналогов классов-"оберток"), а State-нет. Эти паттерны имеют схожие структуры, но решают разные задачи: State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния, в то время как Bridge разделяет абстракцию от ее реализации так, что их можно изменять независимо друг от друга.

Реализация паттерна State основана на паттерне Strategy. Различия заключаются в их назначении.

# Структура проекта

Написав текст файла CMake в п. 3.1 и нажав на кнопку “Configure” (рис. 4), мы запустили настройку параметров сборки нашего проекта (IDE: Visual Studio 14 (2015)).

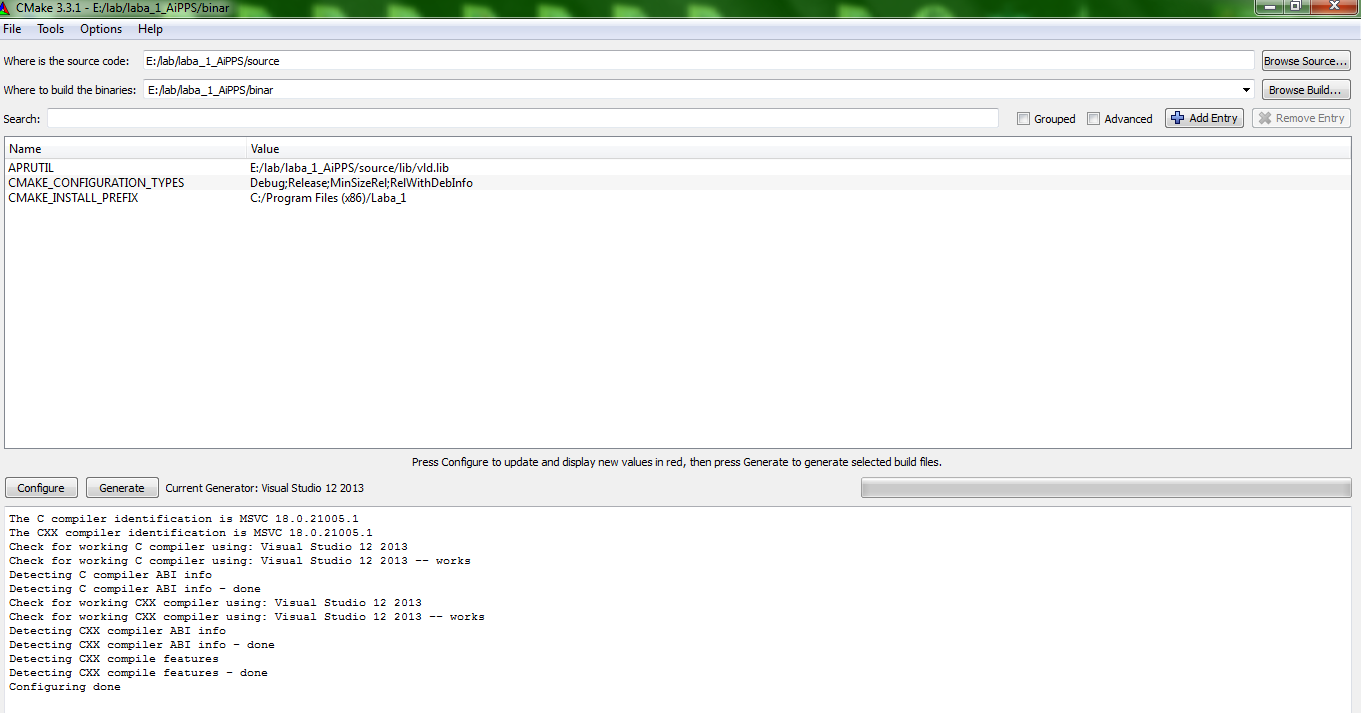


Рисунок 4 Конфигурация проекта

В соответствии с дополнительным заданием, мы подключили к проекту внешнюю библиотеку vld.lib (Visual Leaks Detector - библиотека для отслеживания утечек памяти в C++). Она находится в папке “lib”, как показано на рисунке 5.

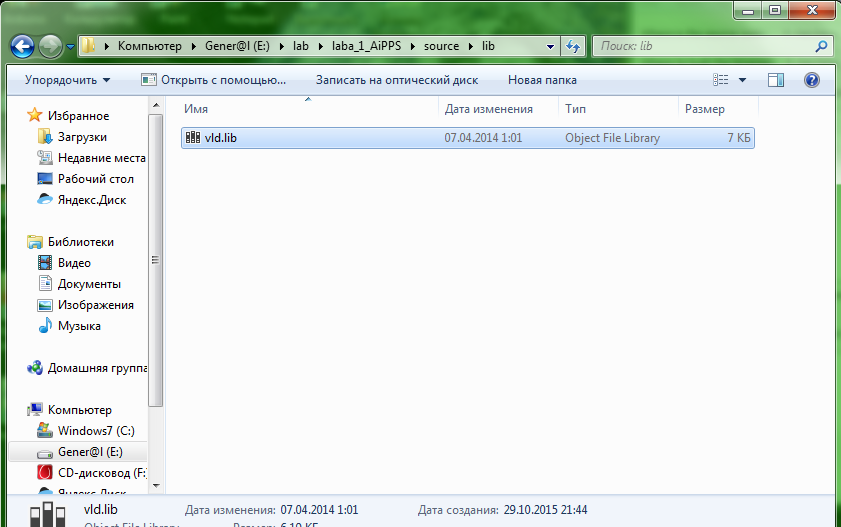


Рисунок 5 Внешняя библиотека

После сборки проекта в CMake содержимое папки, предназначенной для проектных файлов будет выглядеть так, как показано на рисунке 6.

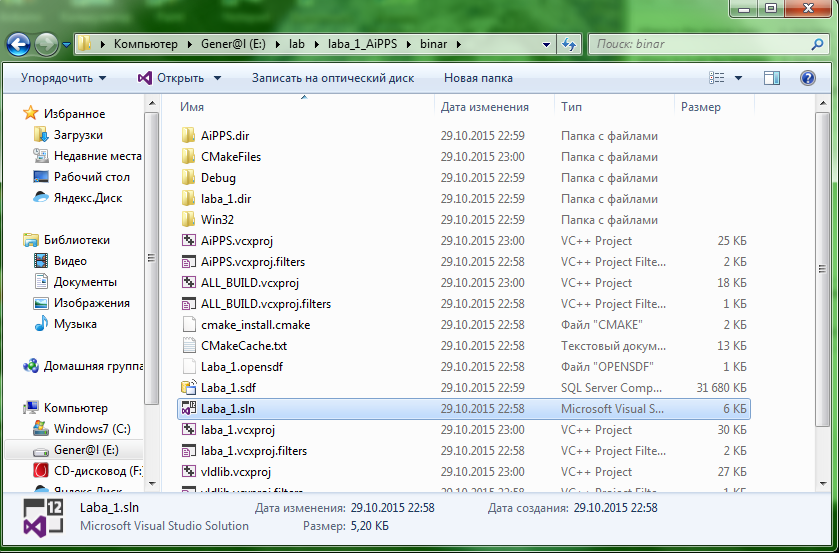


Рисунок 6 Файлы проекта

Исполняемый файл генерируется самой средой разработки по имеющимся файлам решения “\*.sln” и других необходимых файлов для сборки. Исполняемый файл изображен на рисунке 7.

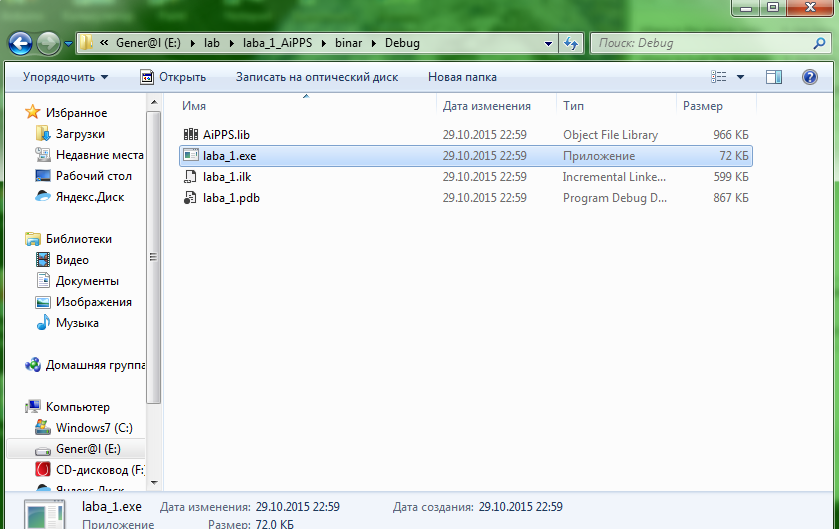


Рисунок 7 Исполняемый файл проекта

Файл “CMakeLists.txt”, в котором указываются все параметры сборки проекта, находится в папке “source” (рис. 8).

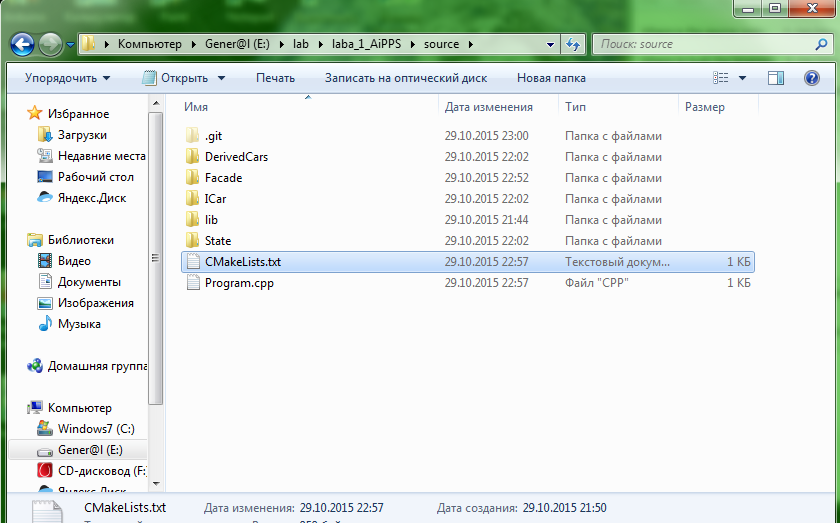


Рисунок 8 CMakeLists.txt

## **Файлы CMake**

CMakeLists.txt

cmake\_minimum\_required(VERSION 3.3)

project(Laba\_1)

set(SOURCES Program.cpp ../source/DerivedCars/BMW.cpp

../source/DerivedCars/VAZ2115.cpp

../source/Facade/Cars.cpp

../source/ICar/ICar.cpp

../source/State/IState.cpp)

set(HEADERS ../source/DerivedCars/BMW.h

../source/DerivedCars/VAZ2115.h

../source/Facade/Cars.h

../source/ICar/ICar.h

../source/State/IState.h)

add\_library(AiPPS STATIC ${SOURCES} ${HEADERS})

include\_directories(../source/DerivedCars Facade ICar State lib)

find\_library(APRUTIL NAMES "vld" PATHS ${PROJECT\_SOURCE\_DIR}/lib)

add\_library(vldlib SHARED ${APRUTIL})

SET\_TARGET\_PROPERTIES(vldlib PROPERTIES LINKER\_LANGUAGE C++)

add\_executable(laba\_1 ${SOURCES} ${HEADERS})

target\_link\_libraries(laba\_1 AiPPS)

# Листинг программы

IState.h

#pragma once

#include "ICar.h"

// Базовый класс

class IState

{

public:

IState(){};

virtual ~IState() {};

virtual void Start(ICar\* iCar) = 0;

virtual void Stop(ICar\* iCar) = 0;

virtual void Speed(ICar\* iCar, int speed) = 0;

protected:

const char\* ENGINE\_ALREADY\_STOPPED = "Двигатель уже остановлен!";

const char\* UNABLE\_TO\_CHANGE\_SPEED = "Нельзя изменить скорость, когда двигатель остановлен!";

const char\* ENGINE\_ALREADY\_STARTED = "Двигатель уже запущен!";

const char\* SET\_SPEED\_TO\_ZERO = "Прежде чем остановить двигатель уменьшите скорость до 0!";

const char\* SPEED\_IS\_NEGATIVE = "Значение скорости не может быть отрицательным!";

};

// опережающие объявления

class MotorStoppedState;

class MotorWorksState;

// Класс для автоматического разрушения одиночек

class SingletonDestroyer

{

public:

~SingletonDestroyer();

void Init(MotorStoppedState\* p);

void Init(MotorWorksState\* p);

private:

// одиночки (классы, наследующие IState)

MotorStoppedState\* ms\_p\_Instance;

MotorWorksState\* mv\_p\_Instance;

};

// Класс состояния "Мотор заглушен"

class MotorStoppedState : public IState

{

public:

void Start(ICar\* iCar);

void Stop(ICar\* iCar);

void Speed(ICar\* iCar, int speed);

public:

static MotorStoppedState\* GetInstance();

protected:

MotorStoppedState() { }

MotorStoppedState(const MotorStoppedState&);

MotorStoppedState& operator=(MotorStoppedState&);

~MotorStoppedState() { };

friend class SingletonDestroyer;

private:

static MotorStoppedState\* \_Instance;

static SingletonDestroyer \_Destroyer;

};

// Класс состояния "Мотор заведен"

class MotorWorksState : public IState

{

public:

void Start(ICar\* iCar);

void Stop(ICar\* iCar);

void Speed(ICar\* iCar, int speed);

public:

static MotorWorksState\* GetInstance();

protected:

MotorWorksState() { }

MotorWorksState(const MotorWorksState&);

MotorWorksState& operator=(MotorWorksState&);

~MotorWorksState() { };

friend class SingletonDestroyer;

private:

static MotorWorksState\* \_Instance;

static SingletonDestroyer \_Destroyer;

};

IState.cpp

#include "IState.h"

SingletonDestroyer::~SingletonDestroyer()

{

if (ms\_p\_Instance != nullptr)

delete ms\_p\_Instance;

if (mv\_p\_Instance != nullptr)

delete mv\_p\_Instance;

}

void SingletonDestroyer::Init(MotorStoppedState\* p)

{

ms\_p\_Instance = p;

}

void SingletonDestroyer::Init(MotorWorksState\* p)

{

mv\_p\_Instance = p;

}

MotorStoppedState\* MotorStoppedState::\_Instance = nullptr;

SingletonDestroyer MotorStoppedState::\_Destroyer;

void MotorStoppedState::Start(ICar\* iCar)

{

iCar->SwitchMotor();

iCar->ChangeState(MotorWorksState::GetInstance());

}

void MotorStoppedState::Stop(ICar\* iCar)

{

// нельзя остановить уже остановленный двигатель

throw exception(ENGINE\_ALREADY\_STOPPED);

}

void MotorStoppedState::Speed(ICar\* iCar, int speed)

{

// нельзя изменить скорость машины с незаведенным двигателем

throw exception(UNABLE\_TO\_CHANGE\_SPEED);

}

MotorStoppedState\* MotorStoppedState::GetInstance()

{

// создаем Singleton

if (!\_Instance)

{

\_Instance = new MotorStoppedState();

\_Destroyer.Init(\_Instance);

}

return \_Instance;

}

MotorWorksState\* MotorWorksState::\_Instance = nullptr;

SingletonDestroyer MotorWorksState::\_Destroyer;

void MotorWorksState::Start(ICar\* iCar)

{

// нельзя запустить уже запущенный двигатель

throw exception(ENGINE\_ALREADY\_STARTED);

}

void MotorWorksState::Stop(ICar\* iCar)

{

// если машина не едет, глушим мотор

if (iCar->GetSpeed() == 0)

{

iCar->SwitchMotor();

iCar->ChangeState(MotorStoppedState::GetInstance());

}

else

{

throw exception(SET\_SPEED\_TO\_ZERO);

}

}

void MotorWorksState::Speed(ICar\* iCar, int speed)

{

// если значение скорости корректно (>=0), устанавливаем его

if (speed >= 0)

iCar->SetSpeed(speed);

else

{

throw exception(SPEED\_IS\_NEGATIVE);

}

}

MotorWorksState\* MotorWorksState::GetInstance()

{

// создаем Singleton

if (!\_Instance)

{

\_Instance = new MotorWorksState();

\_Destroyer.Init(\_Instance);

}

return \_Instance;

}

ICar.h

#pragma once

#include <iostream>

#include <exception>

#include <Windows.h>

using namespace std;

class IState; // объявление вместо include

// Базовый класс "Автомобиль"

class ICar

{

public:

ICar();

virtual ~ICar(){};

public:

virtual const int GetSpeed(){ return 0; } // возвращает значение скорости

virtual void StartMotor() = 0; // запускает двигатель

virtual void StopMotor() = 0; // останавливает двигатель

virtual void ChangeSpeed(int speed) = 0; // устанавливает новое значение скорости (для IState)

virtual void SetSpeed(int speed) = 0; // setter для скорости

virtual void ChangeState(IState\* instance) = 0; // изменяет состояние автомобиля

virtual void SwitchMotor() = 0; // изменяет состояние мотора (заведен/заглушен)

protected:

int \_speed; // значение скорости

bool \_motorWorks; // состояние мотора (заведен/заглушен)

// индивидуальные характеристики автомобиля

int \_motorEnableSpeed; // время запуска двигателя

int \_changeSpeed; // время разгона до нужной скорости

IState\* \_state; // состояние автомобиля

};

ICar.cpp

#include "ICar.h"

ICar::ICar()

{

\_speed = 0;

\_motorWorks = false;

}

BMW.h

#pragma once

#include "ICar.h"

#include "IState.h"

// Класс Автомобиль "BMW"

class BMW : public ICar

{

public:

BMW();

~BMW();

public:

const int GetSpeed(){ return \_speed; }

void StartMotor();

void StopMotor();

void ChangeSpeed(int speed);

void SetSpeed(int speed);

void ChangeState(IState\* instance);

void SwitchMotor();

};

BMW.cpp

#include "BMW.h"

BMW::BMW()

{

\_speed = 0;

\_motorWorks = false;

\_motorEnableSpeed = 1000;

\_changeSpeed = 500;

ChangeState(MotorStoppedState::GetInstance());

}

BMW::~BMW()

{

}

void BMW::StartMotor()

{

\_state->Start(this);

Sleep(\_motorEnableSpeed);

}

void BMW::StopMotor()

{

\_state->Stop(this);

}

void BMW::ChangeSpeed(int speed)

{

\_state->Speed(this, speed);

Sleep(\_changeSpeed);

}

void BMW::SetSpeed(int speed)

{

\_speed = speed;

}

void BMW::ChangeState(IState\* instance)

{

\_state = instance;

}

void BMW::SwitchMotor()

{

\_motorWorks = !\_motorWorks;

}

VAZ2115.h

#pragma once

#include "ICar.h"

#include "IState.h"

// Класс Автомобиль "VAZ 2115"

class VAZ2115 : public ICar

{

public:

VAZ2115();

~VAZ2115();

public:

const int GetSpeed(){ return \_speed; }

void StartMotor();

void StopMotor();

void ChangeSpeed(int speed);

void SetSpeed(int speed);

void ChangeState(IState\* instance);

void SwitchMotor();

};

VAZ2115.cpp

#include "VAZ2115.h"

VAZ2115::VAZ2115()

{

\_speed = 0;

\_motorWorks = false;

\_motorEnableSpeed = 2000;

\_changeSpeed = 1500;

ChangeState(MotorStoppedState::GetInstance());

}

VAZ2115::~VAZ2115()

{

}

void VAZ2115::StartMotor()

{

\_state->Start(this);

Sleep(\_motorEnableSpeed);

}

void VAZ2115::StopMotor()

{

\_state->Stop(this);

}

void VAZ2115::ChangeSpeed(int speed)

{

\_state->Speed(this, speed);

Sleep(\_changeSpeed);

}

void VAZ2115::SetSpeed(int speed)

{

\_speed = speed;

}

void VAZ2115::ChangeState(IState\* instance)

{

\_state = instance;

}

void VAZ2115::SwitchMotor()

{

\_motorWorks = !\_motorWorks;

}

Cars.h

#pragma once

#include "BMW.h"

#include "VAZ2115.h"

// Типы (марки) автомобилей

typedef enum

{

C\_BMW,

C\_VAZ2115

}ECarType;

// Фасадный класс Автомобили

class Cars

{

public:

Cars(ECarType carType);

~Cars();

public:

void SetCar(ECarType carType);

int GetSpeed(){ return \_ICar->GetSpeed(); }

void StartMotor();

void StopMotor();

void SetSpeed(int speed);

private:

void ChangeState(IState\* iState);

void SwitchMotor();

private:

// указатели на производные классы

BMW\* \_BMW;

VAZ2115\* \_VAZ2115;

ICar\* \_ICar; // указатель на базовый класс

protected:

const char\* MACHINE\_BMW = "Это BMW";

const char\* MACHINE\_VAZ2115 = "Это VAZ-2115";

const char\* MOTOR\_STARTING = "Мотор заводится...";

const char\* MOTOR\_STARTED = "Мотор заведен";

const char\* MOTOR\_STOPPED = "Мотор заглушен";

const char\* CHANGING\_SPEED = "Меняем скорость...";

const char\* NEW\_SPEED\_SET = "Установлена новая скорость: ";

const char\* SPEED\_MEASURE = " км/ч";

};

Cars.cpp

#include "Cars.h"

Cars::Cars(ECarType carType)

{

\_BMW = 0;

\_VAZ2115 = 0;

if (carType == ECarType::C\_BMW)

{

\_BMW = new BMW();

\_ICar = \_BMW;

cout << MACHINE\_BMW << endl;

}

else if (carType == ECarType::C\_VAZ2115)

{

\_VAZ2115 = new VAZ2115();

\_ICar = \_VAZ2115;

cout << MACHINE\_VAZ2115 << endl;

}

}

Cars::~Cars()

{

if (\_BMW)

delete \_BMW;

if (\_VAZ2115)

delete \_VAZ2115;

}

void Cars::SetCar(ECarType carType)

{

if (carType == ECarType::C\_BMW)

{

if (\_BMW)

delete \_BMW;

\_BMW = new BMW();

\_ICar = \_BMW;

cout << MACHINE\_BMW << endl;

}

else if (carType == ECarType::C\_VAZ2115)

{

if (\_VAZ2115)

delete \_VAZ2115;

\_VAZ2115 = new VAZ2115();

\_ICar = \_VAZ2115;

cout << MACHINE\_VAZ2115 << endl;

}

}

void Cars::StartMotor()

{

cout << MOTOR\_STARTING << endl;

\_ICar->StartMotor();

cout << MOTOR\_STARTED << endl;

}

void Cars::StopMotor()

{

\_ICar->StopMotor();

cout << MOTOR\_STOPPED << endl;

}

void Cars::SetSpeed(int speed)

{

cout << CHANGING\_SPEED << endl;

\_ICar->ChangeSpeed(speed);

cout << NEW\_SPEED\_SET << speed << SPEED\_MEASURE << endl;

}

void Cars::ChangeState(IState\* iState)

{

\_ICar->ChangeState(iState);

}

void Cars::SwitchMotor()

{

\_ICar->SwitchMotor();

}

Program.cpp

/\* Реализация возможностей паттернов State, Facade, Singleton

Выполнено Дианой Белик\*/

#include "Cars.h"

#include <locale>

int main()

{

setlocale(LC\_ALL, "Rus");

Cars\* cars = new Cars(ECarType::C\_BMW);

try

{

cars->StartMotor();

cars->SetSpeed(150);

cars->SetSpeed(0);

cars->StopMotor();

cars->SetCar(ECarType::C\_VAZ2115);

cars->StartMotor();

cars->SetSpeed(80);

cars->SetSpeed(0);

cars->StopMotor();

}

catch (exception ex)

{

cout << ex.what() << endl;

}

delete cars;

system("pause");

return 0;

}