Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Алтайский государственный технический университет им. И. И. Ползунова»

Факультет информационных технологий
Кафедра прикладной математики

Отчет защищен с оценкой_____
Преподаватель_____С. М. Старолетов
(подпись) (и. о., фамилия)

«_____»_____2017 г.
дата

Комплексный отчет

по дисциплине «Архитектурное проектирование и паттерны программирования»

ЛР 09.03.04.07.000О

Студент группы ПИ-42 Д. С. Гутяр
(и. о., фамилия)

Преподаватель доцент, к.ф.-м.н. С. М. Старолетов
должность, ученое звание (и. о., фамилия)

Барнаул 2017

Постановка задачи

Предметная область программы

Приложение представляет собой двумерную игру, где игрок может управлять разными объектами: человеком, танком, автомобилем и вертолетом на сцене. Сцена представляет собой прямоугольную область в центре экрана, по которой могут перемещаться объекты. Каждый объект имеет свои оригинальные характеристики, особенности и территорию действия. Так человек, автомобиль и танк могут перемещаться только по земле (территория в нижней части экрана), а вертолет летает в верхней части экрана. При этом автомобиль движется быстрее, чем человек и танк, человек может прыгать вверх, а вертолет может перемещаться по двум осям координат одновременно, тогда как танк или автомобиль - только по прямой. Есть вариации людей: обычный, с джетпаком, зацепившийся за вертолет. Обычный человек может только бегать и прыгать, двое других – летать, а отличаются они представлением на экране.

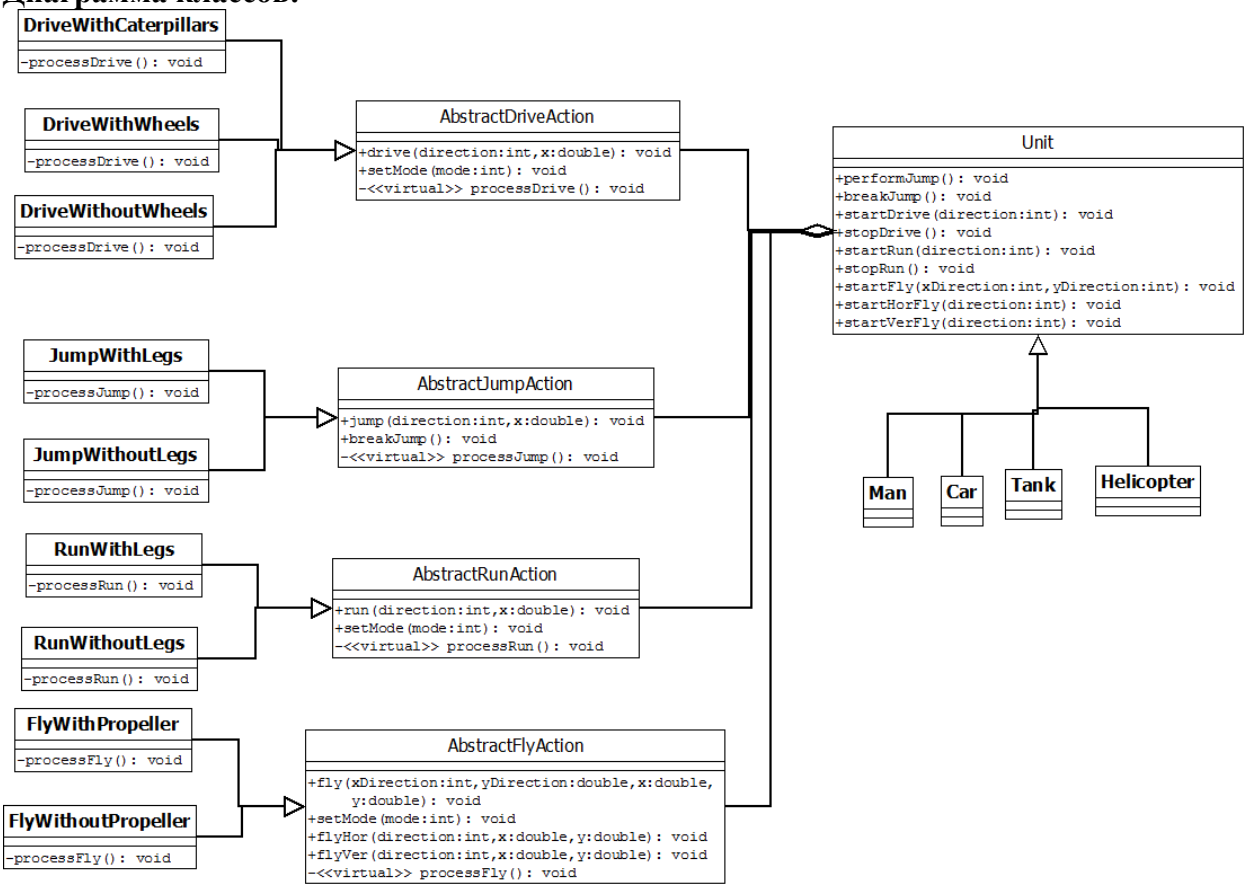
Используемые паттерны

Делегирование

Назначение:

Класс объекта на сцене Unit делегирует выполнение поданных ему команд классу действия, наследуемому от одного из абстрактных классов: AbstractDriveAction, AbstractFlyAction, AbstractRunAction, AbstractJumpAction. Такая архитектура позволяет изменять функциональные возможности класса Unit просто заменяя указатель на одно действие другим, наследуемым от общего для них интерфейса (абстрактного класса). Процесс расширения возможностей класса сводится к созданию новых классов действий, при этом имеющийся код изменяется минимально (только для внедрения экземпляров новых классов)

Диаграмма классов:



Исходный код:
unit.h

```
class Unit:public ObserverAction,public SubjectUnit,public ImageUnit
{
public:
   Unit();
   Unit(ObserverUnit *sc, int k);
   ~Unit();
   virtual void performJump();//выполнение действия прыжка
   virtual void breakJump();//прекращение действия прыжка
   virtual void startDrive(int xDirection);//начать езду в направлении
   virtual void stopDrive();//прекратить езду
   virtual void startRun(int xDirection);//начать бег в направлении
   virtual void stopRun();//прекратить бег
   virtual void startHorFly(int xMotionMode);//начать горизонтальный полет в направлении
   virtual void stopHorFly();//прекратить горизонтальный полет
   virtual void startVerFly(int yMotionMode);//начать вертикальный полет
   virtual void stopVerFly();//прекратить вертикальный полет
   void stopFly();//прекратить горизонтальный и вертикальный полет
//...
protected:
   AbstractDriveAction *driveAction;//экземпляры классов-делегатов
   AbstractFlyAction *flyAction;
   AbstractJumpAction *jumpAction;
   AbstractRunAction *runAction;
};

class Car:public Unit
{
public:
   static const int HEIGHT=30;//высота объекта
   static const int WIDTH=50;//ширина объекта
   static Car* getInstance(ObserverUnit *sc, int k);//получить экземпляр класса
private:
   Car(ObserverUnit *sc, int k);
   static Car* instance;
};

class Helicopter:public Unit
{
public:
   static const int WIDTH=200;
   static const int HEIGHT=100;
   Helicopter(ObserverUnit *sc, int k);
};

class Tank:public Unit
{
public:
   static const int WIDTH=100;
   static const int HEIGHT=50;
   Tank(ObserverUnit *sc, int k);
};

class Man:public Unit
```

```
{
public:
   static const int WIDTH=20;
   static const int HEIGHT=30;
   Man(ObserverUnit *sc, int k,Man* man=NULL);
   void setFlyAction(AbstractFlyAction* flyAct);//функции для изменения
   void setDriveAction(AbstractDriveAction* driveAct);//действий объекта
   void setRunAction(AbstractRunAction* runAct);
   void setJumpAction(AbstractJumpAction* jumpAct);
   void performJump();
private:
   Man* nextMan;
};
```

unit.cpp

```
#include "unit.h"

Unit::Unit()
{}

Unit::Unit(ObserverUnit *sc, int k):tempSprait(0),direction(MOTIONLEFT)
{
   attach(sc);
   key=k;
}

Unit::~Unit()
{}

void Unit::startDrive(int xDirection)
{
   stopFly();
   stopRun();
   driveAction->drive(xPos,xDirection);
}

void Unit::stopDrive()
{
   driveAction->setMode(MOTIONNONE);
}

void Unit::startRun(int xDirection)
{
   stopFly();
   stopDrive();
   runAction->run(xPos,xDirection);
}

void Unit::stopRun()
{
   runAction->setMode(MOTIONNONE);
}

void Unit::startHorFly(int yDirection)
{
```

```
    stopDrive();
    stopRun();
    flyAction->flyHor(xPos,yPos,yDirection);
}

void Unit::stopHorFly()
{
    flyAction->setHorMode(MOTIONNONE);
}

void Unit::startVerFly(int yMotionMode)
{
    breakJump();
    flyAction->flyVer(xPos,yPos,yMotionMode);
}

void Unit::stopVerFly()
{
    flyAction->setVerMode(MOTIONNONE);
}

void Unit::stopFly()
{
    stopHorFly();
    stopVerFly();
}

void Unit::performJump()
{
    stopVerFly();
    jumpAction->jump(yPos);
}

void Unit::breakJump()
{
    jumpAction->breakJump();
}

Car* Car::instance=NULL;

Car::Car(ObserverUnit *sc, int k):Unit(sc,k)
{
    shootAction=new ShootWithoutGun(this);
    driveAction=new DriveWithWheels(this);
    flyAction=new FlyWithoutPropeller(this);
    jumpAction=new JumpWithoutLegs(this);
    runAction=new RunWithoutLegs(this);
    width=WIDTH;
    height=HEIGHT;
    area=sc->getSceneRect();
    area.setBottom((area.bottom()-height)*0.9);
    area.setRight(area.right()-width);
    setActionArea(area);
    direction=MOTIONLEFT;
    xPos=area.right();
```

```
    yPos=area.bottom();
}

Car* Car::getInstance(ObserverUnit *sc, int k)
{
    if(instance==NULL)
        instance=new Car(sc,k);
    return instance;
}

Helicopter::Helicopter(ObserverUnit *sc, int k):Unit(sc,k)
{
    shootAction=new ShootByRocket(this);
    driveAction=new DriveWithoutWheels(this);
    flyAction=new FlyWithPropeller(this);
    jumpAction=new JumpWithoutLegs(this);
    runAction=new RunWithoutLegs(this);
    width=WIDTH;
    height=HEIGHT;
    area=sc->getSceneRect();
    area.setBottom((area.bottom()-height)*0.4);
    area.setRight(area.right()-width);
    area=area;
    setActionArea(area);
    direction=MOTIONLEFT;
    xPos=area.right();
    yPos=area.top()*1.1;
}

Tank::Tank(ObserverUnit *sc, int k):Unit(sc,k)
{
    shootAction=new ShootByShell(this);
    driveAction=new DriveWithCaterpillars(this);
    flyAction=new FlyWithoutPropeller(this);
    jumpAction=new JumpWithoutLegs(this);
    runAction=new RunWithoutLegs(this);
    width=WIDTH;
    height=HEIGHT;
    area=sc->getSceneRect();
    area.setBottom((area.bottom()-height)*0.9);
    area.setRight(area.right()-width);
    area=area;
    setActionArea(area);
    direction=MOTIONLEFT;
    xPos=area.right();
    yPos=area.bottom();
}

Man::Man(ObserverUnit *sc, int k, Man *man):Unit(sc,k)
{
    shootAction=new ShootWithoutGun(this);
    driveAction=new DriveWithoutWheels(this);
    flyAction=new FlyWithoutPropeller(this);
    jumpAction=new JumpWithLegs(this);
    runAction=new RunWithLegs(this);
```

```
    width=WIDTH;
    height=HEIGHT;
    area=sc->getSceneRect();
    area.setBottom((area.bottom()-height)*0.9);
    area.setRight(area.right()-width);
    area=area;
    setActionArea(area);
    direction=MOTIONRIGHT;
    xPos=area.left();
    yPos=area.bottom();
    spraitSize=8;
    nextMan=man;
}

void Man::setDriveAction(AbstractDriveAction *driveAct)
{
    driveAction=driveAct;
}

void Man::setFlyAction(AbstractFlyAction *flyAct)
{
    flyAction=flyAct;
}

void Man::setRunAction(AbstractRunAction *runAct)
{
    runAction=runAct;
}

void Man::setJumpAction(AbstractJumpAction *jumpAct)
{
    jumpAction=jumpAct;
}

void Man::performJump()
{
    stopVerFly();
    jumpAction->jump(yPos);
    if(nextMan!=NULL)
        nextMan->performJump();
}
```

unitactions.h

```
class AbstractDriveAction
{
public:
    AbstractDriveAction(ObserverAction *un);
    void drive(double x,int motionMode);//начать ехать
    void setMode(int motionMode);//установить режим
};

class AbstractFlyAction
{
public:
    AbstractFlyAction(ObserverAction *un);
```

```
    void fly(double x,double y,int xMotionMode,int yMotionMode);//начать полет
    void flyVer(double x,double y,int yMotionMode);//начать вертикальный полет
    void flyHor(double x,double y,int xMotionMode);//начать горизонтальный полет
    void setMode(int xMotionMode,int yMotionMode); //установить режим
};

class AbstractRunAction
{
public:
    AbstractRunAction(ObserverAction *un);
    void run(double x, int motionMode); //начать бег
    void setMode(int motionMode);//установить режим
};

class AbstractJumpAction{
public:
    AbstractJumpAction(ObserverAction *un);
    void jump(double y);//совершить прыжок
    void breakJump();//прервать прыжок
```

unitactions.cpp

```
#include "unitactions.h"

void AbstractDriveAction::drive(double x, int motionMode)
{
    bool actived;
    actived=timer->isActive();
    setMode(motionMode);
    setPos(x);
    if(!actived)
    {
        timer->start();
        processDrive();
    }
}

void AbstractFlyAction::fly(double x, double y, int xMotionMode, int yMotionMode)
{
    bool actived;
    actived=timer->isActive();
    setMode(xMotionMode,yMotionMode);
    setPos(x,y);
    if(!actived)
    {
        timer->start();
        processFly();
    }
}

void AbstractFlyAction::flyVer(double x, double y, int yMotionMode)
{
    fly(x,y,xMode,yMotionMode);
}

void AbstractFlyAction::flyHor(double x, double y, int xMotionMode)
```

```
{
   fly(x,y,xMotionMode,yMode);
}

void AbstractJumpAction::jump(double y)
{
   setPos(y);
   if(!timer->isActive())
   {
      initial();
      timer->start();
      processJump();
   }
}

void AbstractJumpAction::breakJump()
{
   timer->stop();
}

void AbstractRunAction::run(double x,int motionMode)
{
   bool actived;
   actived=timer->isActive();
   setMode(motionMode);
   setPos(x);
   if(!actived)
   {
      timer->start();
      processRun();
   }
}

void AbstractRunAction::setMode(int motionMode)
{
   mode=motionMode;
   if(mode==MOTIONNONE)//остановить движение
      timer->stop();
   initial();
}
```

behavior.h

```
class DriveWithWheels:public AbstractDriveAction{
public:
   DriveWithWheels(ObserverAction *un);

};

class DriveWithCaterpillars:public AbstractDriveAction{
public:
   DriveWithCaterpillars(ObserverAction *un);
};

class DriveWithoutWheels:public AbstractDriveAction{
public:
```

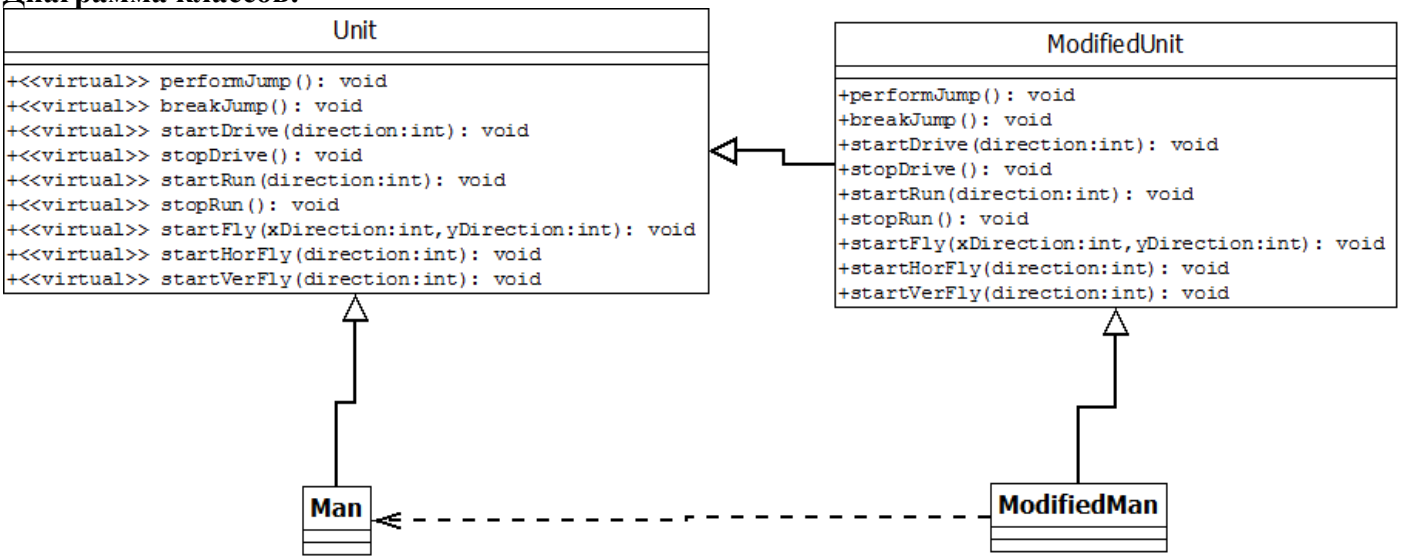
```
    DriveWithoutWheels(ObserverAction *un);
};

class RunWithLegs:public AbstractRunAction{
public:
    RunWithLegs(ObserverAction *un);
};

class RunWithoutLegs:public AbstractRunAction{
public:
    RunWithoutLegs(ObserverAction *un);
};

class FlyWithPropeller:public AbstractFlyAction{
public:
    FlyWithPropeller(ObserverAction *un);
};

class FlyWithoutPropeller:public AbstractFlyAction{
public:
    FlyWithoutPropeller(ObserverAction *un);
};

class JumpWithLegs:public AbstractJumpAction{
public:
    JumpWithLegs(ObserverAction *un);
};

class JumpWithoutLegs:public AbstractJumpAction{
public:
    JumpWithoutLegs(ObserverAction *un);
};
```

Декоратор

Назначение:

Для изменения поведения абстрактного класса Unit используется класс ModifiedUnit, который декорирует его и позволяет реализовать несвойственное поведение. От него наследуется класс ModifiedMan, который декорирует поведение класса Man. Функции абстрактного класса в ModifiedUnit перегружены таким образом, чтобы сначала реализовывать его поведение, а потом поведение «обернутого» класса.

Диаграмма классов:



Исходный код:
modifiedunit.h

```
class ModifiedUnit : public Unit,public ObserverUnit
{
public:
    ModifiedUnit(ObserverUnit *sc, int k);
    virtual void drawing(QGraphicsScene *sc,QHash<int,ContextStruct> spraits)=0;//отрисовка
    void performJump();//совершить прыжок
    void startDrive(int xDirection);//начать езду
    void stopDrive();//прервать езду
    void startHorFly(int xMotionMode);//начать горизонтальный полет
    void stopHorFly();//прервать горизонтальный полет
    void startVerFly(int yModtionMode);//начать вертикальный полет
    void stopVerFly();//прервать вертикальный полет
    void startRun(int xDirection);//начать бег
    void stopRun();//прервать бег
protected:
    double offsetLeftX,offsetRightX,offsetY;
    Unit *unit;
};

class ModifiedMan:public ModifiedUnit
{
public:
    const int WIDTH=10;
    const int HEIGHT=20;
    ModifiedMan(ObserverUnit *sc, int k);
    ModifiedMan(ObserverUnit *sc, Man *un);
    void drawing(QGraphicsScene *sc, QHash<int, ContextStruct> images);
```

```
};
```

modifiedunit.cpp

```
#include "modifiedunit.h"

ModifiedUnit::ModifiedUnit(ObserverUnit *sc, int k):Unit(sc,k)
{}

void ModifiedUnit::performJump()
{
   Unit::performJump();
   unit->performJump();
}

void ModifiedUnit::startDrive(int xDirection)
{
   Unit::startDrive(xDirection);
   unit->startDrive(xDirection);
}

void ModifiedUnit::stopDrive()
{
   Unit::stopDrive();
   unit->stopDrive();
}

void ModifiedUnit::startHorFly(int xMotionMode)
{
   Unit::startHorFly(xMotionMode);
   unit->startHorFly(xMotionMode);
}

void ModifiedUnit::stopHorFly()
{
   Unit::stopHorFly();
   unit->stopHorFly();
}

void ModifiedUnit::startVerFly(int yMotionMode)
{
   Unit::startVerFly(yMotionMode);
   unit->startVerFly(yMotionMode);
}

void ModifiedUnit::stopVerFly()
{
   Unit::stopVerFly();
   unit->stopVerFly();
}

void ModifiedUnit::startRun(int xDirection)
{
   Unit::startRun(xDirection);
   unit->startRun(xDirection);
}
```

```
void ModifiedUnit::stopRun()
{
   Unit::stopRun();
   unit->stopRun();
}

ModifiedMan::ModifiedMan(ObserverUnit *sc, int k):ModifiedUnit(sc,k)
{
   runAction=new RunWithoutLegs(this);
   driveAction=new DriveWithoutWheels(this);
   shootAction=new ShootWithoutGun(this);
   flyAction=new FlyWithPropeller(this);
   jumpAction=new JumpWithoutLegs(this);
   width=WIDTH;
   height=HEIGHT;
   area=sc->getSceneRect();
   area.setBottom((area.bottom()-height)*0.9);
   area.setRight(area.right()-width);
   area=area;
   setActionArea(area);
   direction=MOTIONRIGHT;
   xPos=area.left();
   yPos=area.bottom();
   offsetLeftX=Man::WIDTH/2;
   offsetRightX=0;
   offsetY=0;
   unit=new Man(this,KEYIMAGEMAN);
}

ModifiedMan::ModifiedMan(ObserverUnit *sc, Man *un):ModifiedMan(sc,KEYIMAGEMAN)
{
   unit=un;
}

void ModifiedMan::drawing(QGraphicsScene *sc,QHash<int,ContextStruct> images)
{
   double xDraw,yDraw;
   yDraw=yPos+offsetY;
   xDraw=xPos;
   if(direction==MOTIONLEFT)
      xDraw+=offsetLeftX;
   else xDraw+=offsetRightX;
   QPixmap qp(sc->width(),sc->height());
   QPainter pn(&qp);
   QImage img;
   sc->render(&pn);
   img=images.value(key).images.at(tempSprait);
   if(direction!=images[key].direction)
      img=img.mirrored(true,false);
   pn.drawImage(QRect(xDraw,yDraw,width,height),img);
   sc->addPixmap(qp);
   unit->drawing(sc,images);
}
```

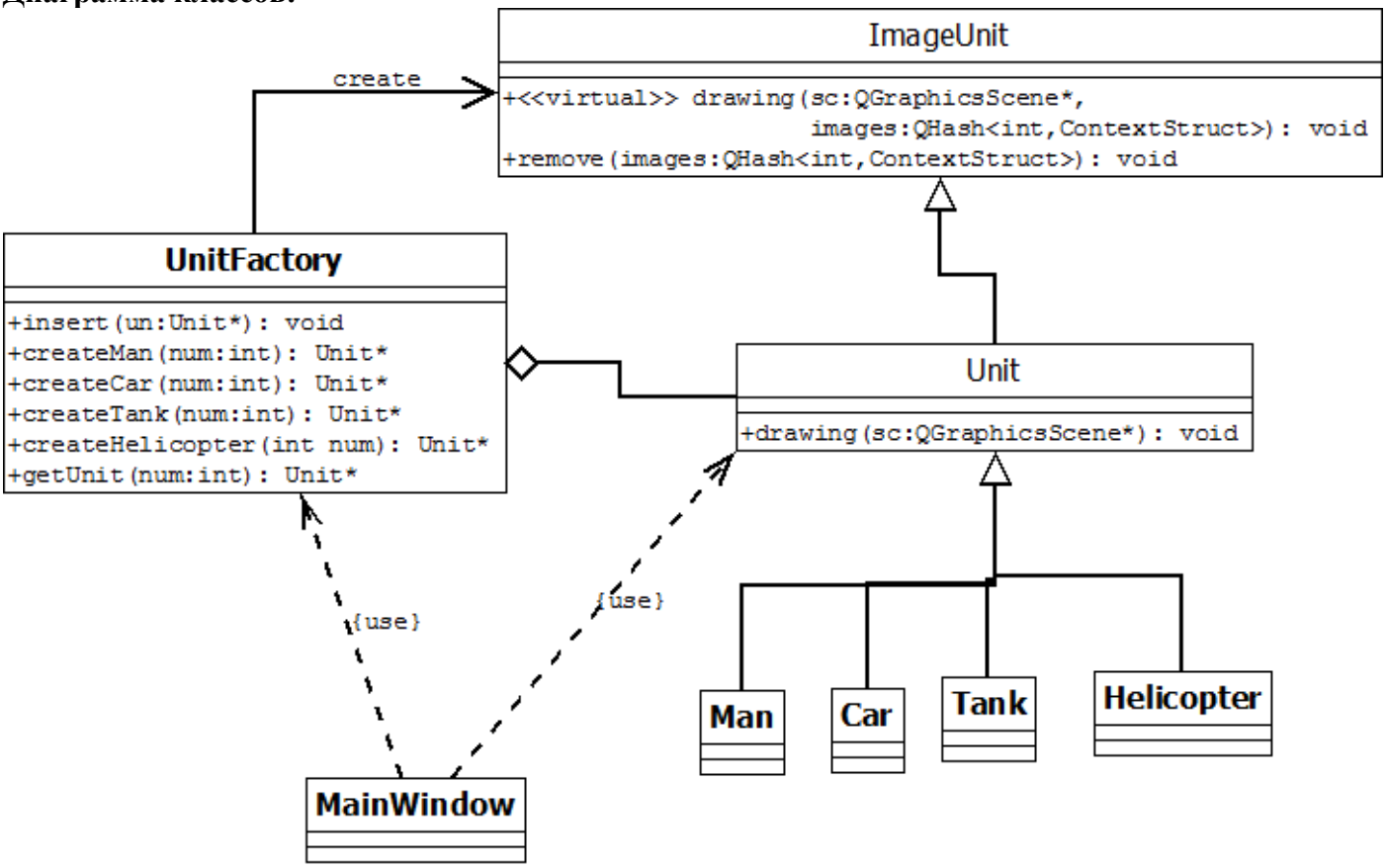
Пример:

Приспособленец

Назначение:

Приспособленец ImageUnit, представляющий объект Unit не хранит некоторые конкретные значения состояния этого объекта. Данный контекст передается ему при непосредственном вызове функций, работающих с ним. Сам контекст хранится в главном окне программы в единственном экземпляре и передается объектам класса Unit по необходимости. К тому же доступ к приспособленцам реализован через фабрику приспособленцев UnitFactory, которая возвращает указатель на приспособленца, если он создан и создает его при отсутствии

Диаграмма классов:



Исходный код:

imageunit.h

```
struct ContextStruct{
    QVector<QImage> images;
    int direction;
```

```
};

class ImageUnit
{
public:
   virtual void drawing(QGraphicsScene *sc,QHash<int,ContextStruct> images)=0;//отрисовка
   virtual void remove(QHash<int, ContextStruct> images);//удаление из общего множества

protected:
   ImageUnit();
   int key;
};
```

imageunit.cpp

```
ImageUnit::ImageUnit()
{}

void ImageUnit::remove(QHash<int, ContextStruct> images)
{
   images.remove(key);
}
```

unitfactory.h

```
class UnitFactory {
public:
   UnitFactory (ObserverUnit *sc) ;
   ~UnitFactory ();
   void insert(Unit* un);//добавление нового объекта к фабрике
   virtual Unit* createMan (int num) ;//вернуть или создать объект
   virtual Unit *createCar(int num) ;
   virtual Unit* createTank (int num) ;
   virtual Unit* createModMan (int num) ;
   virtual Unit *createHelicopter(int num);
   virtual Unit *getUnit(int num);//вернуть созданный объект
private:
   QVector<Unit*> units;
   ObserverUnit *scene;
   Man* lastMan;
};
```

unitfactory.cpp

```
Unit *UnitFactory::getUnit(int num)
{
   if(num<units.size())
      return units.at(num);
   return NULL;
}

Unit *UnitFactory::createMan(int num)
{
   if(num<units.size())
      return units.at(num);
   else
```

```
    {
        Man *unit=new Man(scene,KEYIMAGEMAN,lastMan);
        lastMan=unit;
        units<<unit;
        return unit;
    }
}

Unit* UnitFactory::createCar(int num)
{
    if(num<units.size())
        return units.at(num);
    else
    {
        Car *unit=Car::getInstance(scene,KEYIMAGECAR);
        units<<unit;
        return unit;
    }
}

Unit *UnitFactory::createTank(int num)
{
    if(num<units.size())
        return units.at(num);
    else
    {
        Tank *unit=new Tank(scene,KEYIMAGETANK);
        units<<unit;
        return unit;
    }
}

Unit* UnitFactory::createHelicopter(int num)
{
    if(num<units.size())
        return units.at(num);
    else
    {
        Helicopter *unit=new Helicopter(scene,KEYIMAGEHELIC);
        units<<unit;
        return unit;
    }
}

Unit* UnitFactory::createModMan(int num)
{
    if(num<units.size())
        return units.at(num);
    else
    {
        JetmanBuilder *b1=new JetmanBuilder1(scene);
        ModifiedGenerator *gen=new ModifiedGenerator(b1);
        gen->construct();
        Unit *unit=gen->getResult();
        units<<unit;
```

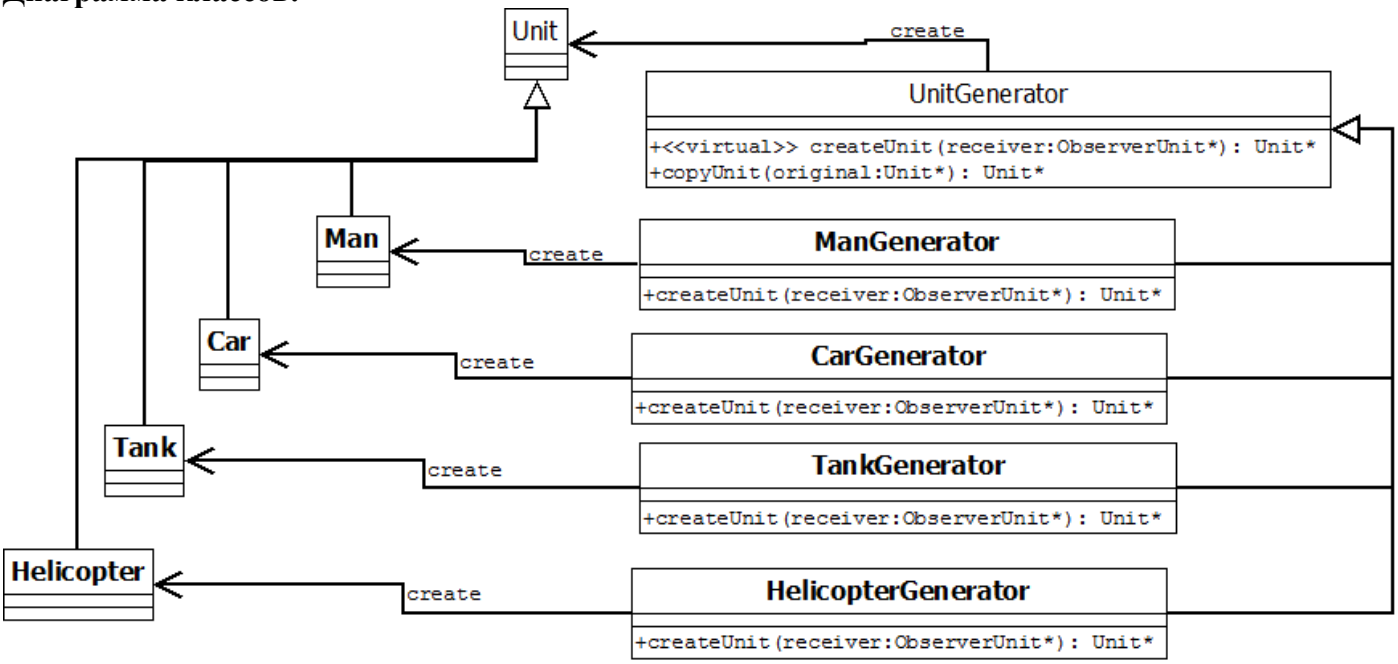
```
        return unit;
    }
}

void UnitFactory::insert(Unit *un)
{
    units<<un;
}
```

Фабричный метод

Назначение:

Строится параллельная иерархия классов, содержащих перегружаемый фабричный метод для создания соответствующих экземпляров классов – объектов сцены. Используется для генерации объектов различного типа, таких, как автомобили, вертолеты, люди и танки. Такая реализация позволяет не взаимодействовать напрямую с объектами при их создании.

Диаграмма классов:



Исходный код:
mangenerator.h

```
class UnitGenerator//абстрактный класс фабрики
{
public:
    UnitGenerator();
    virtual Unit* createUnit(ObserverUnit *receiver)=0;//виртуальный метод создания объектов
    virtual Unit *copyUnit(Unit *original);//перегружаемый метод копирования
private:
    Unit *unit;
};

class ManGenerator:public UnitGenerator//фабрика для создания объектов Man
{
public:
    ManGenerator();
    virtual Man* createUnit(ObserverUnit *receiver);
};
```

```
class CarGenerator:public UnitGenerator//фабрика для создания объектов Car
{
public:
   CarGenerator();
   virtual Car* createUnit(ObserverUnit *receiver);
};

class TankGenerator:public UnitGenerator//фабрика для создания объектов Tank
{
public:
   TankGenerator();
   virtual Tank* createUnit(ObserverUnit *receiver);
};

class HelicopterGenerator:public UnitGenerator//фабрика для создания объектов Helicopter
{
public:
   HelicopterGenerator();
   virtual Helicopter* createUnit(ObserverUnit *receiver);
};
```

mangenerator.cpp

```
UnitGenerator::UnitGenerator()
{}

Unit *UnitGenerator::copyUnit(Unit *original)
{
   return createUnit(original->getObserver());
}

ManGenerator::ManGenerator():UnitGenerator()
{}

Man* ManGenerator::createUnit(ObserverUnit *receiver)
{
   return new Man(receiver,KEYIMAGEMAN);
}

TankGenerator::TankGenerator():UnitGenerator()
{}

Tank* TankGenerator::createUnit(ObserverUnit *receiver)
{
   return new Tank(receiver,KEYIMAGETANK);
}

CarGenerator::CarGenerator():UnitGenerator()
{}

Car *CarGenerator::createUnit(ObserverUnit *receiver)
{
   return Car::getInstance(receiver,KEYIMAGECAR);
}

HelicopterGenerator::HelicopterGenerator():UnitGenerator()
```

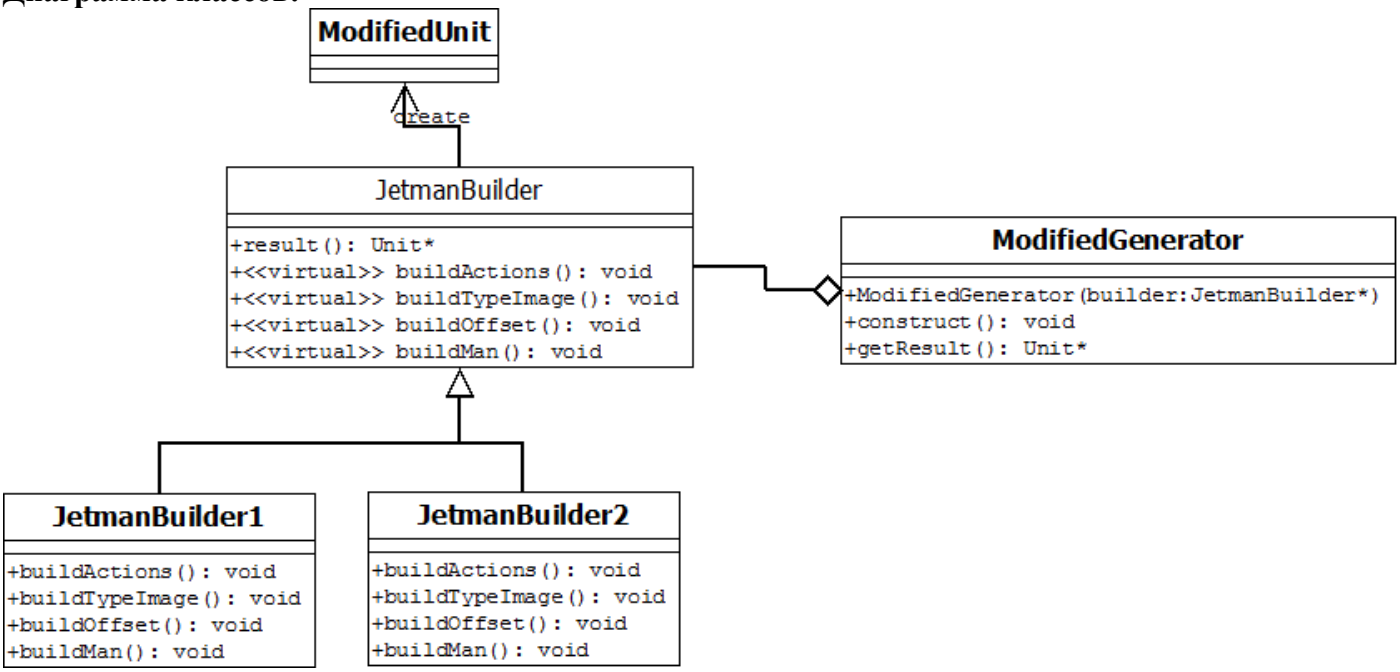
```
{}

Helicopter* HelicopterGenerator::createUnit(ObserverUnit *receiver)
{
   return new Helicopter(receiver,KEYIMAGEHELIC);
}
```

Строитель

Назначение:

Позволяет генерировать объекты определённого типа покомпонентно, управляя процессом сборки извне. В данном случае это абстрактный класс JetmanBuilder, управляемый классом ModifiedGenerator, и два класса наследника, реализующие конкретные варианты сборки: JetmanBuilder1 и JetmanBuilder2. Они позволяют создать сложные объекты в стандартной конфигурации.

Диаграмма классов:



Исходный код:

jetmanbuilder.h

```
class JetmanBuilder
{
public:
   JetmanBuilder(ObserverUnit *sc);
   Unit* result();
   virtual void buildActions()=0;//установить действия для создаваемого объекта
   virtual void buildTypeImage()=0;//установить изображение
   virtual void buildOffset()=0;//установить смещение для изображения
   virtual void buildMan()=0;//установить агрегированный объект
protected:
   ModifiedUnit* buildUnit;
   ObserverUnit *scene;
};

class ModifiedGenerator
{
public:
   ModifiedGenerator(JetmanBuilder *b1);
   void construct();//запустить конструктор
   Unit* getResult();//получить от конструктора результат
private:
   JetmanBuilder *builder;
};

class JetmanBuilder1:public JetmanBuilder
{
```

```
public:
   JetmanBuilder1(ObserverUnit *sc);
   virtual void buildActions();
   virtual void buildTypeImage();
   virtual void buildOffset();
   virtual void buildMan();
private:
};

class JetmanBuilder2:public JetmanBuilder
{
public:
   JetmanBuilder2(ObserverUnit *sc);
   virtual void buildActions();
   virtual void buildTypeImage();
   virtual void buildOffset();
   virtual void buildMan();
private:
};
```

jetmanbuilder.cpp

```
#include "jetmanbuilder.h"

ModifiedGenerator::ModifiedGenerator(JetmanBuilder *b1)
{
   builder=b1;
}

Unit* ModifiedGenerator::getResult()
{
   return builder->result();
}

void ModifiedGenerator::construct()
{
   builder->buildMan();
   builder->buildTypeImage();
   builder->buildActions();
   builder->buildOffset();
}

JetmanBuilder::JetmanBuilder(ObserverUnit *sc)
{
   scene=sc;
}

Unit* JetmanBuilder::result()
{
   return buildUnit;
}

JetmanBuilder1::JetmanBuilder1(ObserverUnit *sc):JetmanBuilder(sc)
{}

void JetmanBuilder1::buildMan()
```

```
{
   Man* man=new Man(scene,KEYIMAGEMAN);
   buildUnit=new ModifiedMan(scene,man);
}

void JetmanBuilder1::buildActions()
{//установить действия и их область для создаваемого объекта
   buildUnit->setRunAction(new RunWithoutLegs(buildUnit));
   buildUnit->setDriveAction(new DriveWithoutWheels(buildUnit));
   buildUnit->setFlyAction(new FlyWithPropeller(buildUnit));
   buildUnit->setJumpAction(new JumpWithoutLegs(buildUnit));
   buildUnit->setActionArea();
}

void JetmanBuilder1::buildOffset()
{
   buildUnit->setOffset(Man::WIDTH/2,0,0);
}

void JetmanBuilder1::buildTypeImage()
{
   buildUnit->setType(KEYIMAGEJETPACK);
}

JetmanBuilder2::JetmanBuilder2(ObserverUnit *sc):JetmanBuilder(sc)
{}

void JetmanBuilder2::buildMan()
{//создать агрегированный объект и установить для него нестандартные действия и их область
   Man* man=new Man(scene,KEYIMAGEMAN);
   man->setRunAction(new RunWithoutLegs(man));
   man->setJumpAction(new JumpWithoutLegs(man));
   man->setActionArea();
   man->setPos(man->getXPos(),man->getYPos());//установить начальную позицию объекта
   buildUnit=new ModifiedMan(scene,man);//передать в создаваемый объект
}

void JetmanBuilder2::buildActions()
{
   buildUnit->setRunAction(new RunWithoutLegs(buildUnit));
   buildUnit->setDriveAction(new DriveWithoutWheels(buildUnit));
   buildUnit->setFlyAction(new FlyWithPropeller(buildUnit));
   buildUnit->setJumpAction(new JumpWithoutLegs(buildUnit));
   buildUnit->setActionArea();
}

void JetmanBuilder2::buildOffset()
{
    buildUnit->setOffset(-Helicopter::WIDTH/2,-Helicopter::WIDTH/2,-
         Helicopter::HEIGHT+Man::HEIGHT/2);
}

void JetmanBuilder2::buildTypeImage()
{//установить тип, размер и ориентацию изображения
   buildUnit->setType(KEYIMAGEHELIC);
```

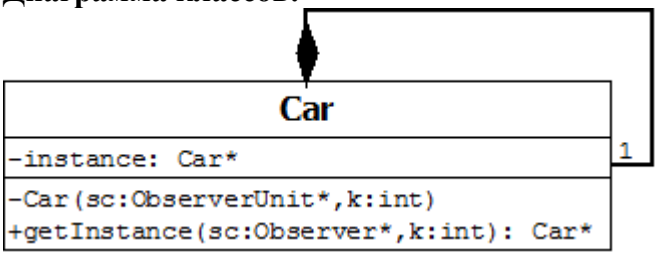
```
    buildUnit->setSize(Helicopter::WIDTH,Helicopter::HEIGHT);
    buildUnit->setDirection(MOTIONRIGHT);
}
```

Синглтон

Назначение:

Этот паттерн позволяет контролировать единственность объекта заданного класса и статическим методом возвращать указатель на него, если он уже был создан. К тому же, в отличии от статического класса, данный объект можно передавать в функцию. В программе таким паттерном служит класс Car.

Диаграмма классов:



Исходный код:

unit.h

```
class Car:public Unit
{
public:
    static const int WIDTH=50;
    static const int HEIGHT=30;
    static Car* getInstance(ObserverUnit *sc, int k);//получить экземпляр класса
private:
    Car(ObserverUnit *sc, int k);//конструктор недоступен извне
    static Car* instance;//статический экземпляр класса
};
```

unit.cpp

```
Car* Car::instance=NULL;//изначально экземпляр не создан

Car::Car(ObserverUnit *sc, int k):Unit(sc,k)
{
    shootAction=new ShootWithoutGun(this);
    driveAction=new DriveWithWheels(this);
    flyAction=new FlyWithoutPropeller(this);
    jumpAction=new JumpWithoutLegs(this);
    runAction=new RunWithoutLegs(this);
    width=WIDTH;
    height=HEIGHT;
    area=sc->getSceneRect();
    area.setBottom((area.bottom()-height)*0.9);
    area.setRight(area.right()-width);
    setActionArea(area);
    direction=MOTIONLEFT;
    xPos=area.right();
    yPos=area.bottom();
}
```

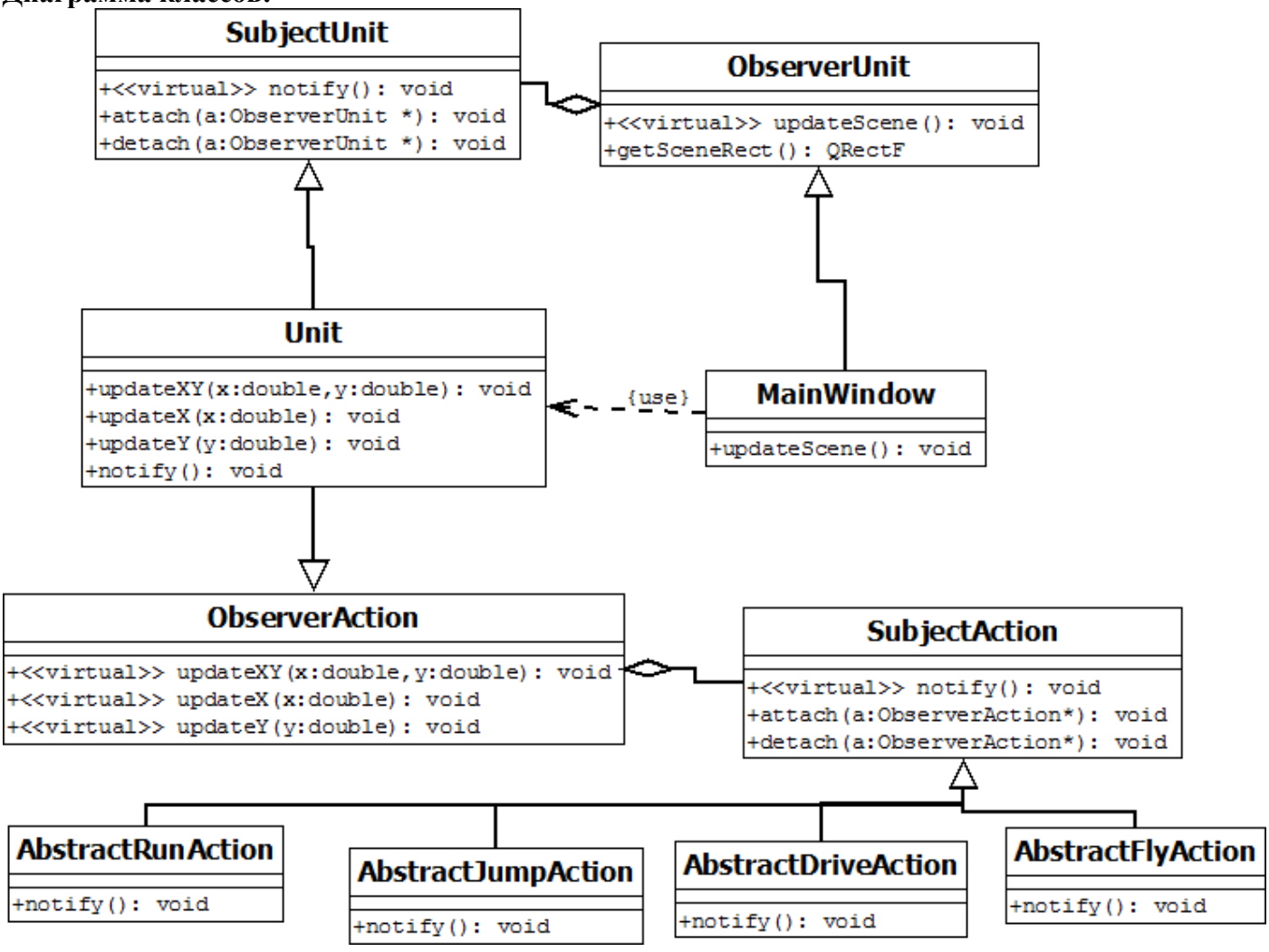
```
Car* Car::getInstance(ObserverUnit *sc, int k)
{
   if(instance==NULL)//если экземпляр еще не был создан
      instance=new Car(sc,k);
   return instance;
}
```

Наблюдатель

Назначение:

Позволяет контролировать состояние объекта, для которого назначаются объекты наблюдатели. Такой объект оповещает наблюдателя об изменении своего состояния. Также наблюдатель может перестать следить за конкретным объектом, вследствие чего он перестанет получать от него информацию. В приложении наследники классов AbstractDriveAction, AbstractRunAction, AbstractJumpAction и AbstractFlyAction могут уведомлять своих наблюдателей об изменении координат. За ними наблюдает класс Unit, который, в свою очередь, является субъектом для класса главного окна и уведомляет его о необходимости отрисовки при изменении координат.

Диаграмма классов:



Исходный код:
obesrver.h
```
class ObserverUnit
{
public:
   virtual void updateScene()=0;//отправить сигнал о необходимости отрисовки
   virtual QRectF getSceneRect()=0;//получить область сцены
};

class SubjectUnit
{
protected:
   ObserverUnit *scene;
public:
   virtual void notify()=0;//отправить сигнал о завершении вычислений
   void attach(ObserverUnit *a);//добавить наблюдателя
   void dettach(ObserverUnit *a);//убрать наблюдателя
};

class ObserverAction
{
public:
   virtual void updateXY(double x,double y)=0;//отправить сигнал об изменении координат
   virtual void updateX(double x)=0;
   virtual void updateY(double y)=0;
};

class SubjectAction
{
protected:
   ObserverAction *unit;
public:
   virtual void notify()=0;
   void attach(ObserverAction *a);
   void dettach(ObserverAction *a);
};
```

observer.cpp
```
#include "observer.h"

void SubjectAction::attach(ObserverAction *a)
{
   unit=a;
}

void SubjectAction::dettach(ObserverAction *a)
{
   if(unit==a)
      unit=NULL;
}

void SubjectUnit::attach(ObserverUnit *a)
{
   scene=a;
}
```

```
void SubjectUnit::dettach(ObserverUnit *a)
{
   if(scene==a)
      scene=NULL;
}
```

unit.h

```
class Unit:public ObserverAction,public SubjectUnit,public ImageUnit
{
public:
   Unit();
   Unit(ObserverUnit *sc, int k);
   ~Unit();
   void updateX(double x);
   void updateY(double y);
   void updateXY(double x, double y);
   void notify();
};
```

unit.cpp

```
void Unit::updateXY(double x, double y)
{
   setPos(x,y);
}

void Unit::updateX(double x)
{
   setX(x);
}

void Unit::updateY(double y)
{
   setY(y);
}

void Unit::notify()
{
   scene->updateScene();
}
```

unitactions.h

```
class AbstractDriveAction: public SubjectAction
{
   Q_OBJECT
public:
   AbstractDriveAction(ObserverAction *un);
   void notify();
};

class AbstractFlyAction: public SubjectAction
{
public:
```

```
    AbstractFlyAction(ObserverAction *un);
    void notify();
};

class AbstractRunAction:public SubjectAction
{
public:
    AbstractRunAction(ObserverAction *un);
    void notify();
};

class AbstractJumpAction: public SubjectAction
{
public:
    AbstractJumpAction(ObserverAction *un);
    void notify();
};
```

mainwindow.h

```
class MainWindow : public QMainWindow,public ObserverUnit
{

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
    void updateScene();
    QRectF getSceneRect();
};
```

mainwindow.cpp

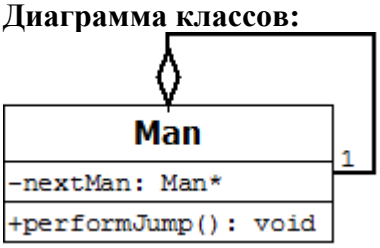
```
//реализация виртуальных методов
void MainWindow::updateScene()
{
    sceneRefresh();
}

QRectF MainWindow::getSceneRect()
{
    return sc->sceneRect();
}
```

Цепочка обязанностей

Назначение:

Используется, когда запрос необходимо обработать последовательно несколькими объеками. Данный паттерн реализуется классом Unit и наследуется классом Man. В реализации класса Man люди выполняют прыжок одновременно, так как запрос на прыжок передается от последнего созданного человека первому.

Диаграмма классов:



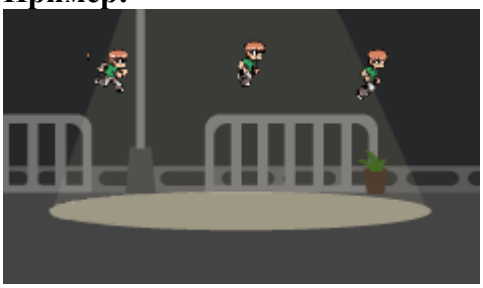
Исходный код:
unit.h

```
class Man:public Unit
{
public:
    static const int WIDTH=20;
    static const int HEIGHT=30;
    Man(ObserverUnit *sc, int k,Man* man=NULL);
    void performJump();//перегрузка базового метода прыжка
private:
    Man* nextMan;
};
```

unit.cpp

```
void Man::performJump()
{
    stopVerFly();
    jumpAction->jump(yPos);
    if(nextMan!=NULL)//передача обязанностей следующему объекту
        nextMan->performJump();
}
```

Пример:



Приложение

Исходные коды программы находятся на прилагаемом диске в каталоге «Исходные коды».