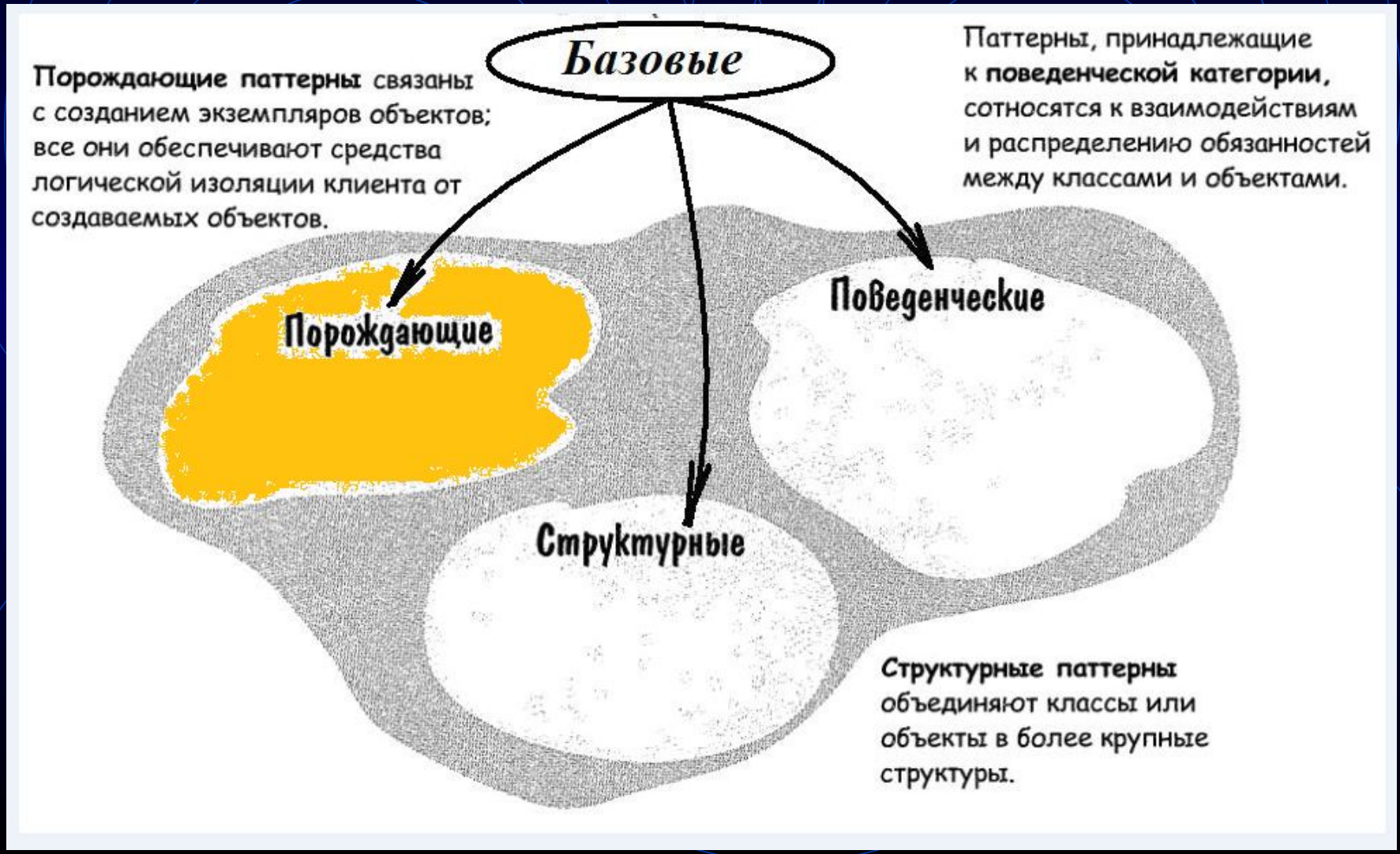
# Тема 3. Порождающие шаблоны проектирования



# Порождающие шаблоны

| | |
|---|---|
| 1. Factory | Простая Фабрика |
| 2. Factory method | Фабричный метод |
| 3. Abstract factory | Абстрактная фабрика |
| 4. Builder | Строитель |
| 5. Prototype | Прототип |
| 6. Singleton | Одиночка |
| 7. Object Pool | Пул объектов |

# Порождающие шаблоны

Порождающие шаблоны позволяют сделать систему независимой от способа создания, композиции и представления объектов.

Они скрывают детали того, как эти классы создаются и стыкуются.

# Порождающие шаблоны

Структура объектов системе не известна.

Единственная информация об объектах, известная системе, - это их интерфейсы, определенные с помощью абстрактных классов.

# Порождающие шаблоны

Порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, *что создается, кто это создает, как и когда.*

*Можно собрать систему из «готовых» объектов* с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

# Порождающие шаблоны

Порождающие паттерны показывают, как сделать дизайн более гибким, хотя и необязательно меньшим по размеру. В частности, их применение позволит легко менять классы, определяющие компоненты системы.

Паттерн, порождающий классы, использует наследование, чтобы варьировать создаваемый класс

Паттерн, порождающий объекты, делегирует создание другому объекту.

# Порождающие шаблоны – краткая характеристика Фабричный метод

Задача; создать игровой лабиринт. Пусть класс CreateMaze (или даже просто функция ) предназначен для создания лабиринта. Если CreateMaze **вызывает виртуальные функции вместо конструкторов для создания** комнат, стен и дверей, то инстанцируемые классы можно подменить, создав подкласс MazeGame и переопределив в нем виртуальные функции.

Такой подход применяется в паттерне **фабричный метод**

# Порождающие шаблоны – краткая характеристика Абстрактная фабрика

Когда функции CreateMaze **в качестве параметра передается объект, используемый для создания** комнат, стен и дверей, то их классы можно изменить, передав другой параметр.

Это пример паттерна **абстрактная фабрика**.

# Порождающие шаблоны – краткая характеристика

## Строитель

Если функции CreateMaze передается объект, способный **целиком создать новый лабиринт с помощью своих операций для добавления** комнат, дверей и стен, можно воспользоваться наследованием для изменения частей лабиринта или способа его построения.

Такой подход применяется в **паттерне строитель.**

# Порождающие шаблоны – краткая характеристика
# Прототип

Если CreateMaze **параметризована прототипами комнаты, двери и стены, которые она затем копирует** и добавляет к лабиринту, то состав лабиринта можно варьировать, заменяя одни объекты-прототипы другими.

Это паттерн **прототип.**

# Порождающие шаблоны – краткая характеристика

## Одиночка

Одиночка может **гарантировать наличие единственного лабиринта** в игре и свободный доступ к нему со стороны всех игровых объектов, не прибегая к глобальным переменным или функциям.

Одиночка также позволяет легко расширить или заменить лабиринт, не трогая существующий код.

# РАЗБИРАЕМ КОНСТРУКЦИЮ ПОРОЖДАЮЩИХ ПАТТЕРНОВ

Начинаем с простой фобрики.

# Простая фабрика

Создает и возвращает объекты запрашиваемого типа, так что все объекты в системе создаются только с помощью фабрики.

Не использует виртуальные функции.

Не делегирует создание объектов другим классам.

Не позволяет строить составные объекты.

# Простая фабрика

*Проблема*

В программе создаются и используются объекты групп взаимосвязанных классов, причем класс объекта динамически выбирается в зависимости от выполнения некоторых условий.

Появляется код типа:

```
if (<условие 1>) myObject = new classObj_1(…);
else if  f (<условие 2>) myObject = new classObj_2(…);
else if  f (<условие 3>) myObject = new classObj_3(…);
…
```

Недостатки такой реализации при внесении изменений:

1 – найти код (а таких позиций в коде может быть много!!)

2 – внести изменения непосредственно в найденный код

*Итог:*

*Код необходимо менять при каждом изменении системы*

# Простая фабрика

*Пример:*

Построить систему автоматического обслуживания клиентов в отеле. Обычно в номере у клиента имеется набор типов завтраков, в каждый из которых клиент может выбрать некоторые ингредиенты.

Доставка завтрака в комнату клиента отеля - это функция

IBreakfast * orderBreakfast(char * name){…}

Завтрак надо собрать, упаковать, доставить в комнату.

# Простая фабрика

```
//Доставка завтрака в комнату клиента отеля
IBreakfast  * orderBreakfast(char * name){
   IBreakfast * breakfast;
   if (strcmp(name,” continent”))  breakfast = new
     continentBreakfast ();
   else  if (strcmp(name,”english”))  breakfast = new
     englishBreakfast ();
   else if (strcmp(name,”american”))  breakfast  = new
     amerBreakfast ();
   breakfast ->  collect();
   breakfast ->  pack();
   return breakfast;
}
```

# Простая фабрика

*Что делать, если в отеле появился новый тип завтрака?*
*Что делать, если нужно отказаться от американского завтрака?*

```
IBreakfast  * orderBreakfast(char * name){
   IBreakfast * breakfast;

   if (strcmp(name,” continent”))  breakfast = new continentBreakfast ();
   else  if (strcmp(name,”english”))  breakfast = new englishBreakfast ();
   else if (strcmp(name,”american”))  breakfast  = new amerBreakfast ();

   breakfast ->  collect();
   breakfast ->  pack();
   return breakfast;
}
```

В КОД НАДО ВНОСИТЬ ИЗМЕНЕНИЯ!

# Простая фабрика

*Выделили операцию создания объекта в метод create() класса Factory*

```
IBreakfast  * orderBreakfast(char * name){
   IBreakfast * breakfast;
   Factory * factory= new Factory();
   breakfast = factory->create(name);
   breakfast ->  collect();
   breakfast ->  pack();
   return breakfast;
}
```

# Простая фабрика

**Переносим код создания в метод**

```
IBreakfast * factory->create(name){
IBreakfast * breakfast;

  if (strcmp(name,” continent”))  breakfast = new continentBreakfast ();
  else  if (strcmp(name,”english”))  breakfast = new englishBreakfast ();
  else if (strcmp(name,”american”))  breakfast  = new amerBreakfast ();

return breakfast;
}
```

В КОД МОЖНО ВНОСИТЬ ИЗМЕНЕНИЯ!

# характеристика
# Фабричный метод

Если создатель объектов **вызывает виртуальные функции вместо конструкторов для создания** комнат, стен и дверей, то инстанцируемые классы можно подменить, создав подкласс MazeGame и переопределив в нем виртуальные функции.

Такой подход применяется в паттерне фабричный метод;

# Factory method
# (Фабричный метод)

Проблема

Необходимо обеспечить создание объектов, при этом отсутствует информация о классе этого объекта.

Решение

Делегировать часть функций вспомогательному классу.

*Результат*

*Строится параллельная иерархия классов, содержащих перегружаемый фабричный метод для создания соответствующих экземпляров классов-продуктов. Создание делегируется подклассам, при этом скрыта реализацию для достижения большей гибкости или возможности расширения функциональности*

# Определение Фабричного метода

Шаблон Factory Method дает возможность объекту инициировать создание другого объекта, ничего не зная о классе этого объекта.

Шаблон используется в случаях:

1) класс заранее не знает, какие объекты необходимо будет создавать, т.к. возможны варианты реализации;

2) класс спроектирован так, что спецификация порождаемого объекта определяется только в наследниках.

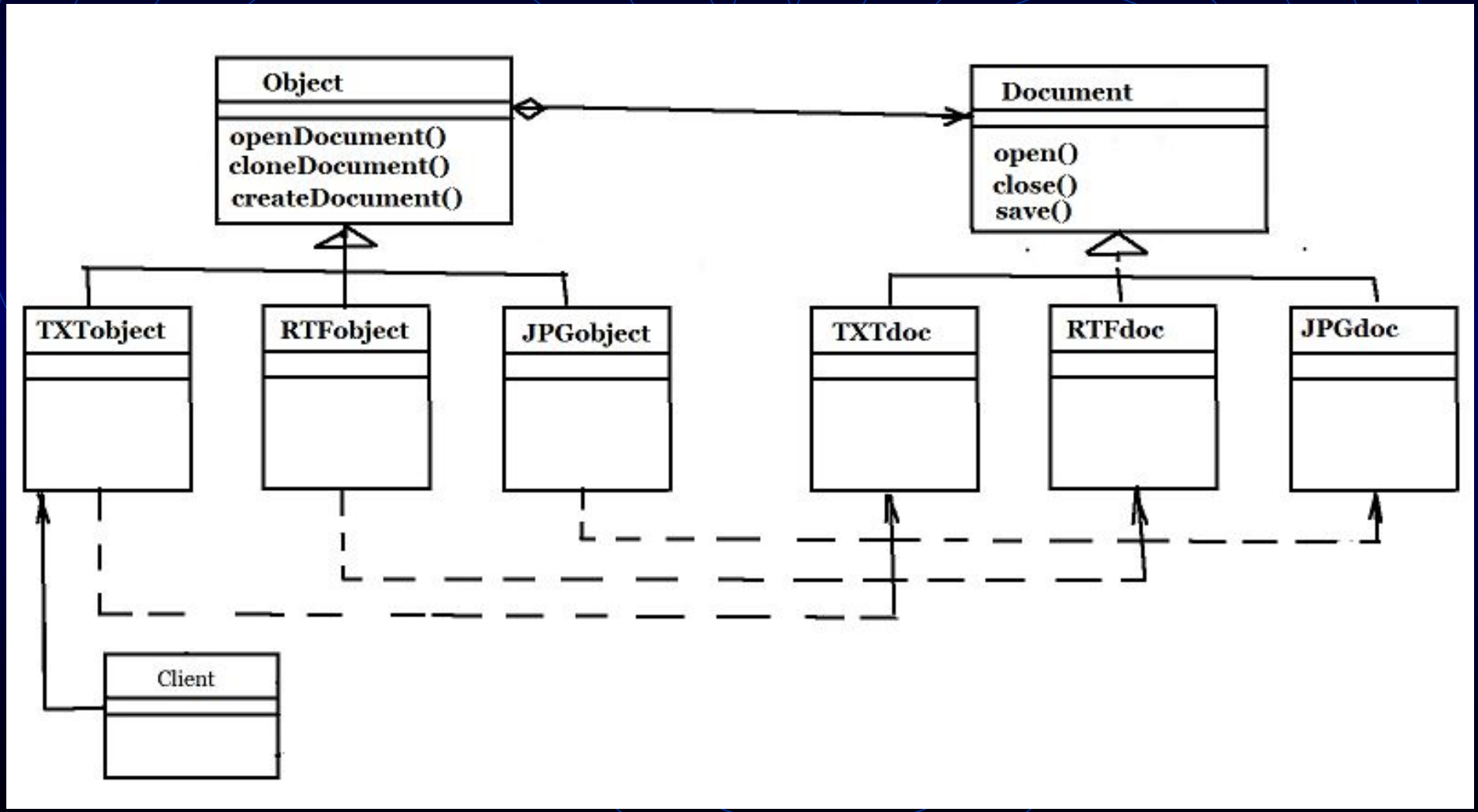
3) класс выделяет и делегирует часть своих функций вспомогательному классу.

# Реализация  Фабричного метода

Строится параллельная иерархия классов, содержащих перегружаемый фабричный метод для создания соответствующих  экземпляров классов-продуктов

Создание делегируется  подклассам, которые решают, какие объекты следует создать

# Реализация Фабричного метода



# Реализация Фабричного метода -1

```
#include <iostream>
using namespace std;

class Document{ // базовый класс
protected:
  char * name;
  int size;
public:
  Document(char* n, int s){name = n; size = s;}
  virtual void display(){cout<< "My name is  "<<
name<<"\n";}
  char * getName(){return name;}
};
```

# Реализация Фабричного метода - 2

```
class TXTdocument : public Document{
private:
  char *data;
public:
   TXTdocument(char* n, int len) : Document(n, len){
       data = new char[len+10];       data[0] = 0;
   }
   TXTdocument(char* n, char * info) : Document(n, sizeof(info)){
      cout<< "\n new TXTdocument from info " << info <<"\n";
int len = strlen(info);
      data = new char[len+10];     memcpy(data, info, len+1);
   }
  virtual void display(){
       cout<< "\nMy name is  "<< name<<"\n";
       cout<< "My text:\n"<< data <<"\n";}
};
```

# Реализация Фабричного метода - 3

```
class JPGdocument : public Document{
private:
  char *data;
public:
JPGdocument(char* n, char * empty) : Document(n, 200){
        data = new char[200];
        char picture[] = "\1\1\1\1\1\1\1\1\n\1\1\1\1\1\1\1\1\n";
        memcpy( data, picture, strlen(picture) + 1 );
    }
   virtual void display(){
        cout<< "\nMy name is  "<< name<<"\n";
        cout<< "My picture:\n"<< data <<"\n";}
};
```

# Реализация Фабричного метода - 4

```
class Object {                  // Класс  Factory method
  private:
  Document * document;
public:
  Object(){}
//Абстрактный метод, реализуемый  наследником,
 // создает и возвращает объект конкретного документа.
virtual  Document * createDocument(char * name, char *
info) = 0;
  Document* cloneDocument(Document* a) {
    return this->createDocument(a-> getName(), "copy");
  }
} ;
```

# Реализация Фабричного метода - 5

```
class TXTobject : public Object{
public:
   TXTobject() : Object() {}
   virtual TXTdocument * createDocument(char * name,
char * info){    return new TXTdocument(name, info); }
};

 class JPGobject : public Object{
public:
   JPGobject() : Object() {}
   virtual JPGdocument * createDocument(char * name,
char * info){   return new JPGdocument(name, info); }
};
```

# Реализация Фабричного метода - 6

```
int main(){
   Object * ft = new TXTobject();
   Document *a= ft->createDocument("document1",
                    "this is a very long text");
   Document *b= ft->createDocument("doc2","short text");
   Document *c= ft->cloneDocument(a);
   a->display();
   b->display();
   c->display();
   Object * fg = new JPGobject();
   Document * g =fg->createDocument("newPictire","");
   g->display();
   return 0;
}
```

# Реализация Фабричного метода – пример работы

```
C:\LECTURES\Patterns\03_Порождающие>FactotyMethod.exe

 new TXTdocument from info this is a very long text

 new TXTdocument from info short text

 new TXTdocument from info copy

My name is  document1
My text:
this is a very long text

My name is  doc2
My text:
short text

My name is  document1
My text:
copy

My name is  newPictire
My picture:
☻☻☻☻☻☻☻☻
☻☻☻☻☻☻☻☻


C:\LECTURES\Patterns\03_Порождающие>_
1Help  2UserMn  3View  4Edit  5Copy  6RenMov  7MkFold  8Delete  9ConfMn  10Quit
```

# Преимущества и недостатки Фабричного метода

**Преимущества**

Избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы.

**Недостатки**

Возникает дополнительный уровень подклассов.

# характеристика Абстрактной фабрики (Известен также под именем Kit)

Создателю объектов **в качестве параметра передается объект, используемый для создания** комнат, стен и дверей, то их классы можно изменить, передав другой параметр.

Это пример паттерна абстрактная фабрика.

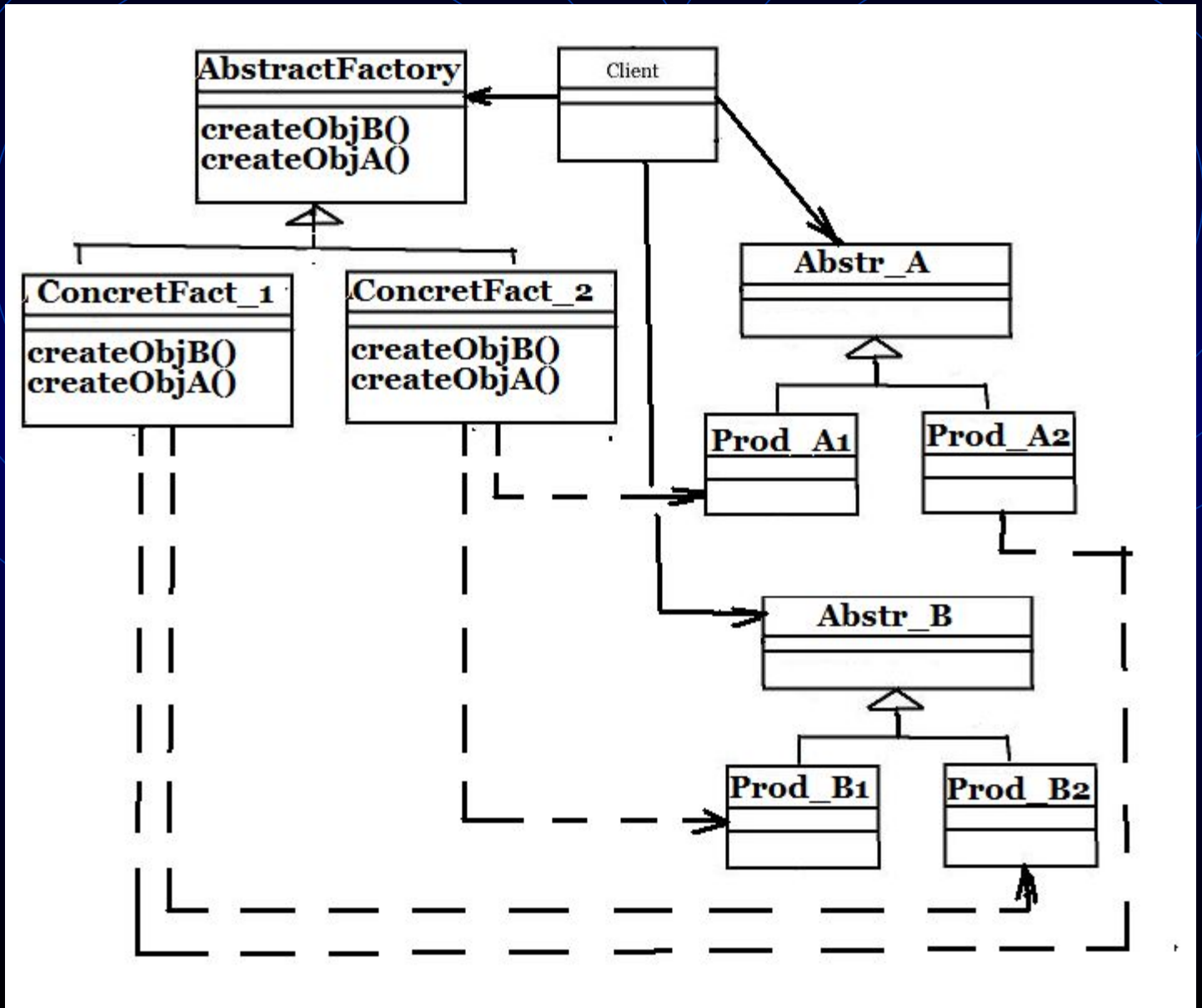
# Абстрактная фабрика (Abstract factory )

Фабрика создает объекты, состоящие из нескольких составных элементов  или обладающие несколькими различными свойствами.

Клиенты создают комплексные объекты или комплексные свойства объектов, пользуясь исключительно интерфейсом AbstractFactory. Клиентам ничего не известно о классах,  реализующих объекты конкретного типа.

Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

# Абстрактная фабрика



# Абстрактная фабрика - пример

Задача

Создать фабрику автомобилей , причем система не должна зависеть от способа создания и реализации входящих в нее объектов – автомобилей

Реализации Абстрактной фабрики и порождаемых ею объектов скрыты от клиента.

# Абстрактная фабрика

```
// интерфейсы и абстрактные классы:
class ICarBody { ... }; // кузов
class  IEngine { ... }; // двигатель
class TCar{ ... };    //  автомобиль с двигателем
                      // и кузовом
class ICarFactory { ... };  // абстрактная фабрика
                            // автомобилей
```

# Абстрактная фабрика

```
//реализация каждого семейства
class SedanCarBody : ICarBody { ... };
                                // кузов седан
class UniversalCarBody : ICarBody { ... } ;
                                // кузов универсал
class DefaultEngine_400 : IEngine { ... } ;
class DefaultEngine_300 : IEngine { ... } ;
```

# Абстрактная фабрика

```
//реализация  Абстрактной фабрики
class MersedesCarFactory : ICarFactory { ... } ;
class FerraryCarFactory : ICarFactory { ... } ;
```

# Абстрактная фабрика

```
class ICarBody { // кузов
protected:
  char  name[20];
  int cost;
  void setName(char * n){
                    memcpy(name, n, strlen(n)+1);}
  void setCost(int c){cost = c; }
public:
  void show(){cout <<"name ="<< name <<endl;}
  ICarBody(){ }
  ~ICarBody(){}
 };
```

# Абстрактная фабрика

```
class  IEngine {     // двигатель
private:
  int power;      int volume;
  void setPower(int p){power = p;}
  void setVolume(int v){volume = v; }
public:
  void show(){cout <<"power ="<< power << "
volume="<<volume<< endl;}
  IEngine (int power, int volume){       setPower(power);
       setVolume(volume);
  }
  ~IEngine(){}
};
```

# Абстрактная фабрика

```
class TCar{   //  автомобиль с двигателем и кузовом
private:
    class  IEngine * engine;
    class ICarBody  * carBody ;
public:
   void show(){engine ->show();  carBody -> show();}
   TCar(ICarBody  * c, IEngine * e ){
                  engine = e; carBody = c; }
   ~TCar(){}
};
```

# Абстрактная фабрика

```
// абстрактная фабрика автомобилей
class ICarFactory {
public:
    virtual TCar * createCar() = 0 ;
};
```

# Абстрактная фабрика

```
//реализация каждого семейства
class UniversalCarBody : ICarBody { // кузов универсал
public:
    UniversalCarBody(){
      setName( "Universal");
      setCost( 178 );
    }
    ~UniversalCarBody(){}
};
```

# Абстрактная фабрика

```
//реализация каждого семейства
class SedanCarBody : ICarBody { // кузов седан
public:
    SedanCarBody(){
      setName( "Sedan");
      setCost( 120 );
    }
    ~SedanCarBody(){}
};
```

# Абстрактная фабрика

```
//реализация каждого семейства

class DefaultEngine_400 : IEngine {
public:
    DefaultEngine_400():IEngine(400, 5000){}
    ~DefaultEngine_400(){}
};

class DefaultEngine_300 : IEngine {
public:
    DefaultEngine_300():IEngine(300, 3500){}
    ~DefaultEngine_300(){}
};
```

# Абстрактная фабрика

```
//реализация  конкретной Абстрактной фабрики
class MersedesCarFactory : ICarFactory {
private:
public:
    virtual TCar * createCar(){
        SedanCarBody * b  = new SedanCarBody();
        DefaultEngine_300 * e = new DefaultEngine_300();
        return new TCar((ICarBody *)b,(IEngine *)e);
    }
    MersedesCarFactory(){}
    ~MersedesCarFactory(){}
};
```

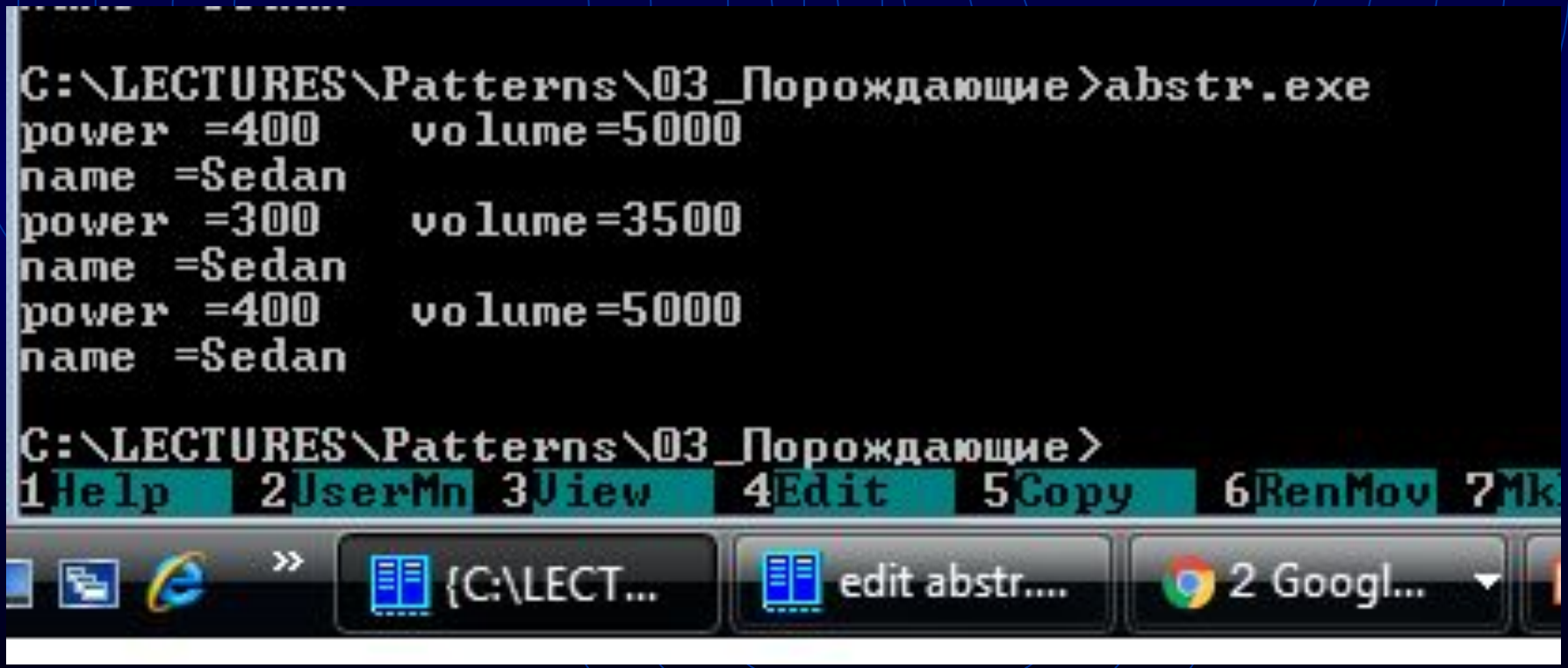
# Абстрактная фабрика

```
//реализация  конкретной Абстрактной фабрики
class FerraryCarFactory : ICarFactory {
private:
public:
    virtual TCar * createCar(){
      SedanCarBody *  b = new SedanCarBody();
      DefaultEngine_400 *  e =  new
DefaultEngine_400();
      return new TCar((ICarBody *)b,(IEngine *)e);
    }
    FerraryCarFactory(){}
    ~FerraryCarFactory(){}
};
```

# Абстрактная фабрика

```
//пример использования в клиенте
int main()  {
class FerraryCarFactory    * af1 = new  FerraryCarFactory ();
class MersedesCarFactory   * af2 = new  MersedesCarFactory ()

    TCar *c1 = af1->createCar();
    c1->show();
    TCar *c2 = af2->createCar();
    c2->show();
    TCar *c3 = af1->createCar();
    c3->show();
    return 0;
}
```

# Абстрактная фабрика



# Абстрактная фабрика

## Полученный результат

Клиентский код использует в работе только интерфейсы.

Реализации Абстрактной фабрики и порождаемых ею объектов скрыты. Такой подход уменьшает зависимости между объектами и повышает гибкость за счет возможности изменения реализаций.

# Абстрактная фабрика – замечания по реализации

Конкретные фабрики можно реализовать с помощью фабричных методов.

# Абстрактная фабрика

Пример

Построить генератор сценариев, в которых есть

1. Место  события (Шахта, Дремучий лес, Ферма)
2. Постоянный обитатель (Шахтер, Фермер)
3. Волшебник / герой (Гном, Рыцарь, Гоблин)

Абстрактная фабрика создает сценарий, делегируя создание отдельных объектов.  Обычная реализация – на основе фабричных методов.

# Преимущества и недостатки Абстрактной фабрики

**Преимущества**

Абстрактная фабрика инкапсулирует ответственность за создание классов и процесс их создания, следовательно, она изолирует клиента от деталей реализации классов. Простая замена Абстрактной фабрики, т.к. она используется в приложении только один раз при инстанцировании.

**Недостатки**

Интерфейс Абстрактной фабрики фиксирует набор объектов, которые можно создать. Расширение Абстрактной фабрики для изготовления новых объектов часто затруднительно.

# Характеристика Прототипа

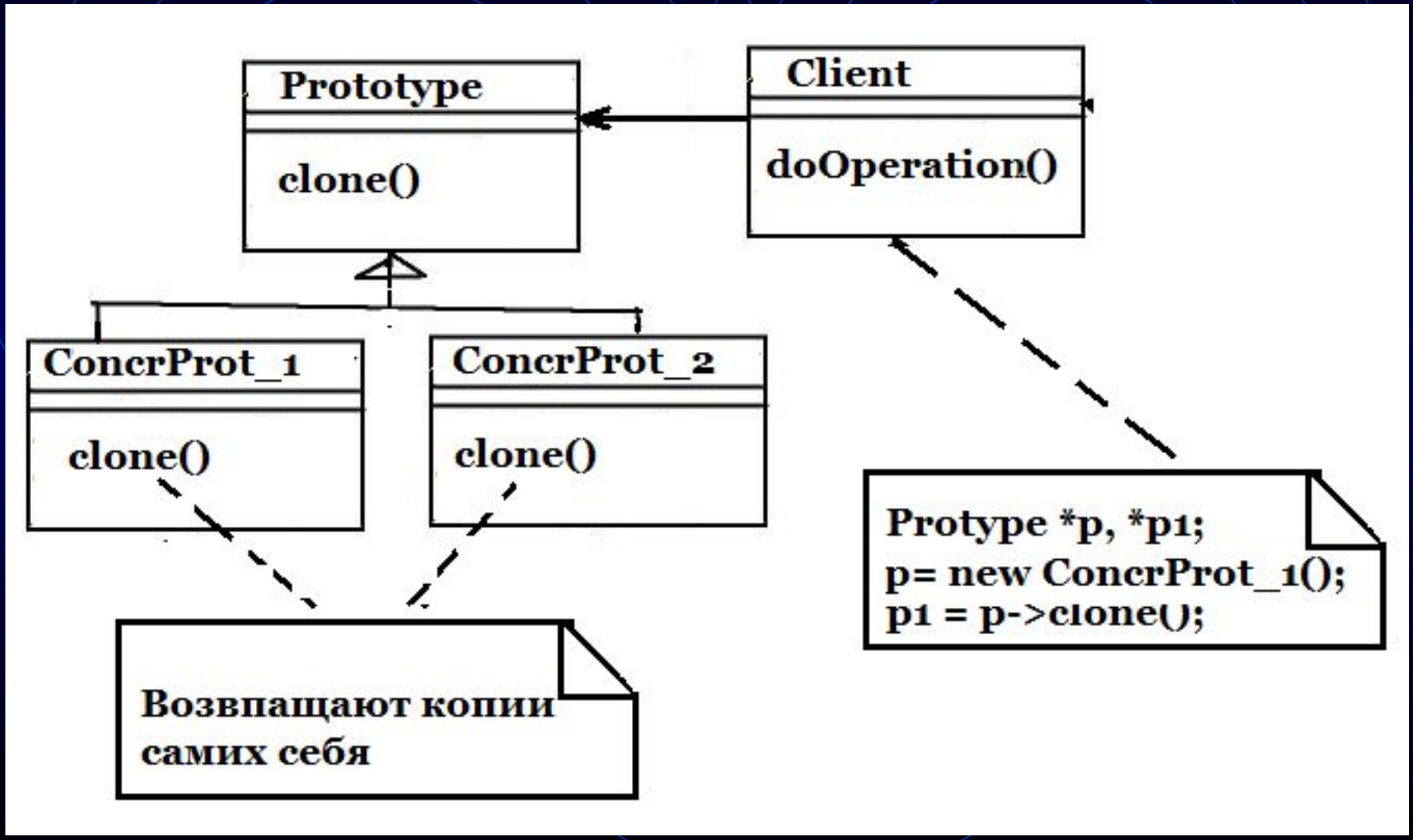
Если CreateMaze **параметризована прототипами комнаты, двери и стены, которые она затем копирует** и добавляет к лабиринту, то состав лабиринта можно варьировать, заменяя одни объекты-прототипы другими.

Это паттерн прототип.

# Прототип (Prototype)

Проблема:
Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа

# Прототип


Prototype
clone()
Client
doOperation()
ConcrProt_1
clone()
ConcrProt_2
clone()
Protype *p, *p1;
p= new ConcrProt_1();
p1 = p->clone();
Возвпащают копии
самих себя

# Прототип – пример (1)

```
class Animal {
protected:
    int age;    char * nameFood;
public:
    Animal() {age = 0; nameFood = new char[20;}
    Animal(int n) {age = n; nameFood = new char[20];}
    Animal(const Animal* animal)     {
        this->age = animal -> age;
        memcpy(this->nameFood,
         animal->nameFood,strlen(animal->nameFood)+1);
    }
    virtual void feed()    {    cout << "Animal age " << age <<
                  " likes to eat  " << nameFood << endl;
    }
    virtual Animal *clone() = 0;
};
```

# Прототип – пример (2)

```
class Cat : public Animal { // кошачьи
public:
    Cat() {memcpy(this->nameFood,"meat",5);}
    Cat(int n):Animal(n) {
     memcpy(this->nameFood,"meat",5);
cout<<"Cat\n";
}
    Animal *clone()     {
        return new Cat(*this);
    }
};
```

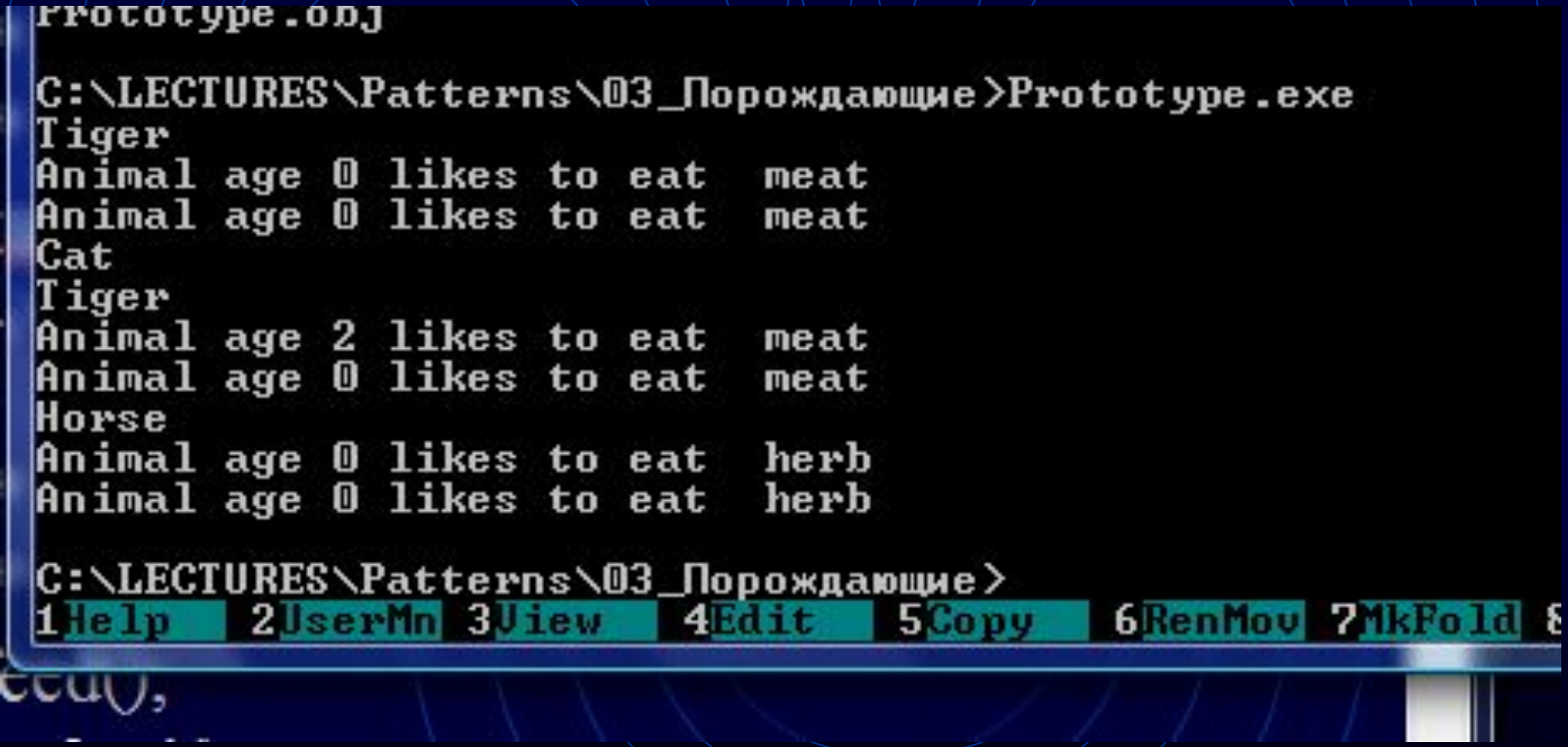
# Прототип – пример (3)

```
class Tiger : public Cat { // тигр
public:
    Tiger() {cout<<"Tiger\n";}
    Tiger(int n):Cat(n) {cout<<"Tiger\n";}

    void growl()    {
        cout << "Tiger growls" << endl;
    }
    Animal *clone()    {
        return new Tiger(*this);
    }
};
```

# Прототип – пример (4)

```
int main(int argc, char *argv[]) {
    Tiger *tigerPrototype = new Tiger();
    Animal *tiger2 = tigerPrototype->clone();
    tigerPrototype ->feed();
    tiger2->feed();
    Animal * newAnimal = new Tiger(2);
    Animal * animal2 = tigerPrototype->clone();
    newAnimal->feed();
    animal2->feed();

    Animal * horse = new Horse();
    Animal * animal3 = horse->clone();
    horse->feed();
    animal3->feed();
    return 0;
}
```

# Прототип - результат работы



# Характеристика Строителя

Если функции CreateMaze передается объект, способный **целиком создать новый лабиринт с помощью своих операций для добавления** комнат, дверей и стен, можно воспользоваться наследованием для изменения частей лабиринта или способа его построения.

Такой подход применяется в паттерне строитель.

# Строитель (**Builder**)

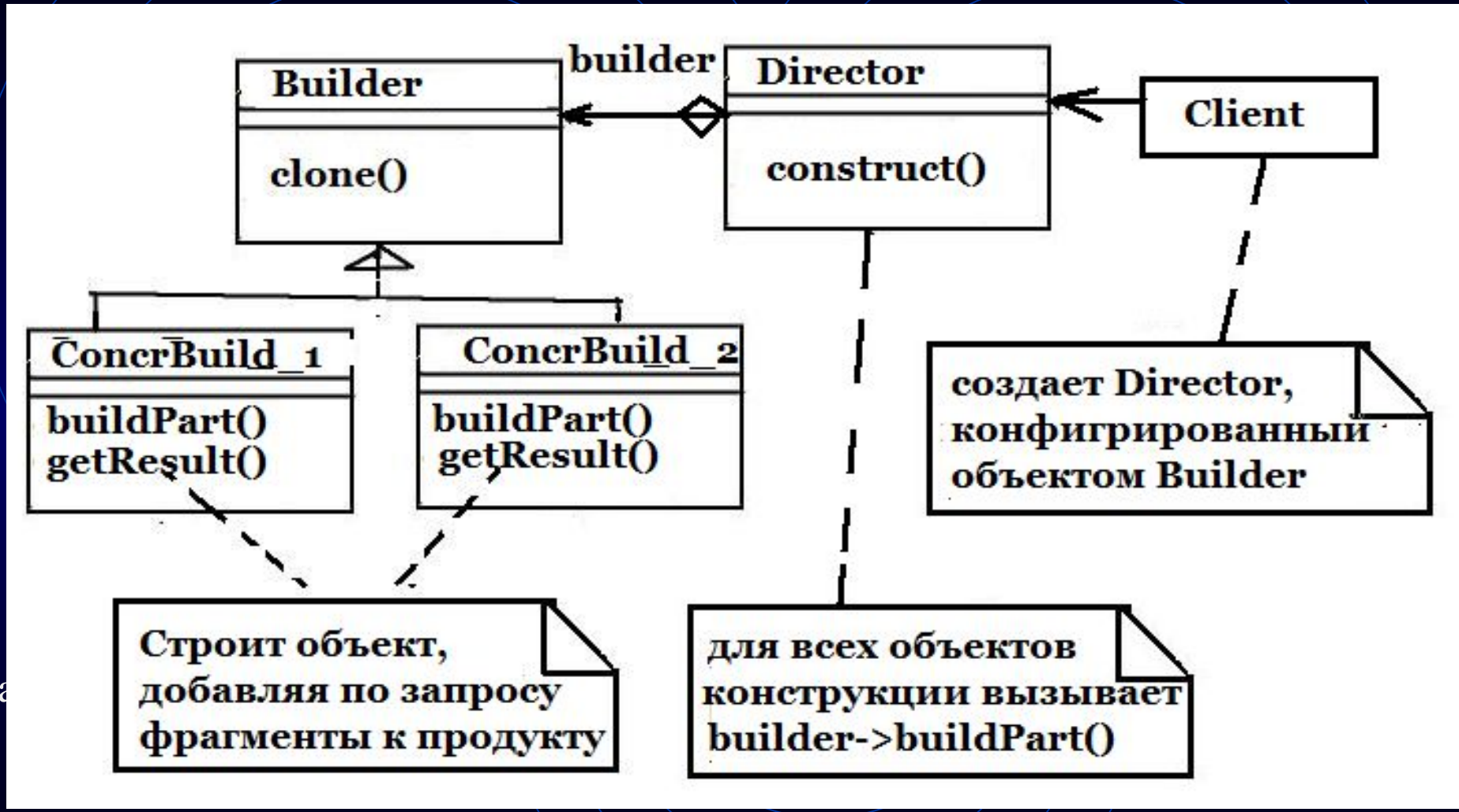
Проблема

Необходимо целиком создать новый объект с помощью своих операций для добавления конструктивных элементов. Можно изменять типы частей формируемого объекта или изменять способ его построения.

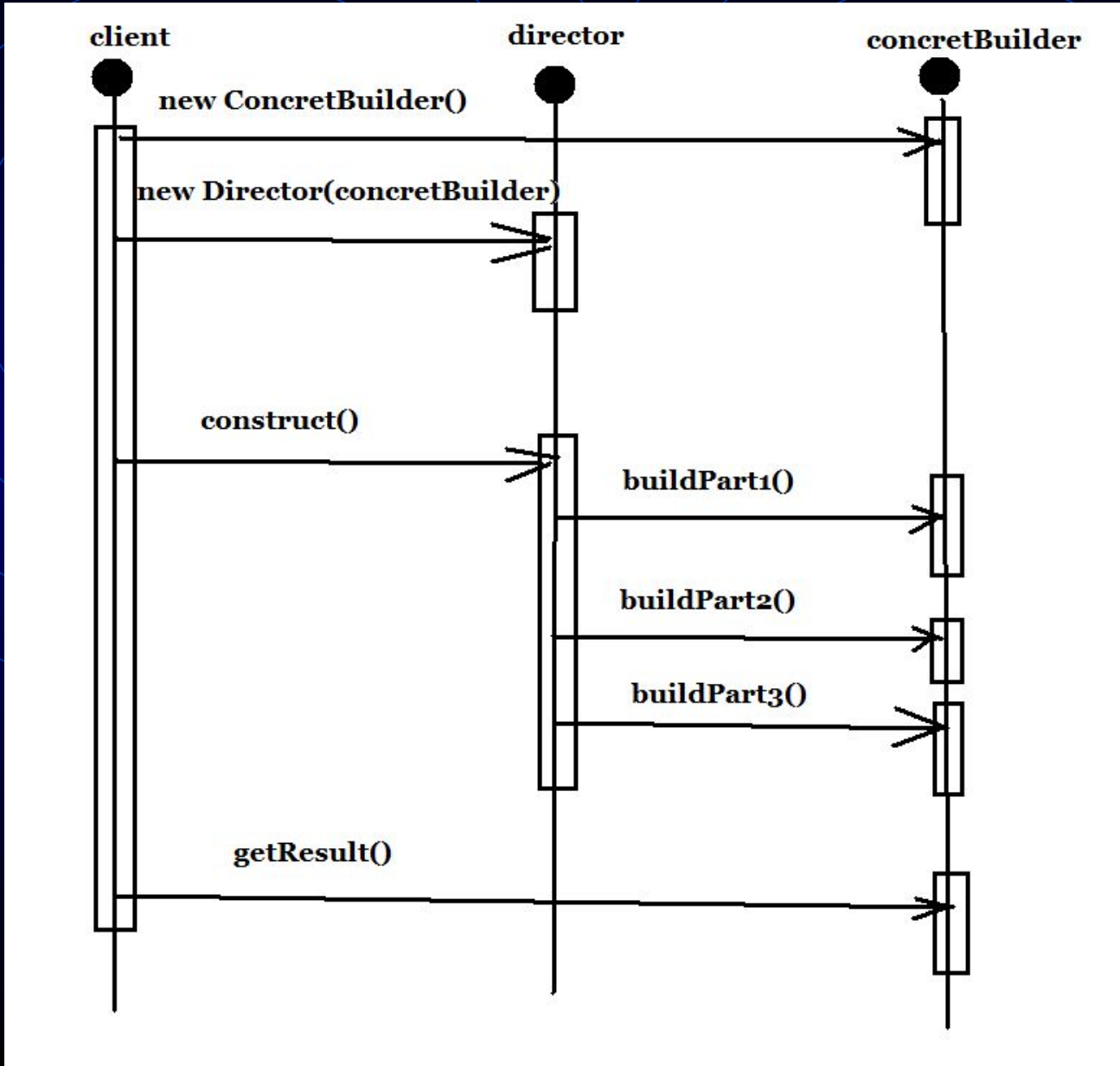
*Результат*

*Отделили конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.*

# Строитель


Builder
clone()
builder
Director
construct()
Client
ConcrBuild_1
buildPart()
getResult()
ConcrBuild_2
buildPart()
getResult()
создает Director, конфигрированный объектом Builder
Строит объект, добавляя по запросу фрагменты к продукту
для всех объектов конструкции вызывает builder->buildPart()

# Строитель (диаграмма последовательности)



# Строитель – пример реализации

Создаем автомат по производству гамбургеров

Конструктор класса

```
Director ::Director (Builder *b){myBuilder = b;}
```

Производство

```
void Director::construct(){
   b-> addSalad();
   b -> addCheese();
   b -> addMeat();
}
```

# Строитель – недостатки

Builder  для класса и создаваемый им продукт жестко связаны между собой, поэтому при внесении изменений в класс продукта скорее всего придется соответствующим образом изменять и класс Builder.

# Строитель и Фабрика

Фабрика используется для создания экземпляра, тип которого зависит от передаваемых параметров.

Строитель используется, когда тип известен, надо лишь по-разному заполнять поля объекта.

Итог: фабрика создает различные типы, для строителя используется один и тот же тип, но с разным наполнением

# Сравнительная характеристика

| **Абстрактная фабрика** | **Фабричный метод** | **Строитель** |
| --- | --- | --- |
| Порождает семейство объектов с определенными интерфейсами. | Порождает один объект с определенным интерфейсом. | Создает в несколько шагов один сложный (составной) объект. |

# Сравнительная характеристика

| **Абстрактная фабрика** | **Фабричный метод** | **Строитель** |
| --- | --- | --- |
| Интерфейс, реализуемый классами. | Метод класса, который переопределяется потомками | Интерфейс строителя, реализуемый классами, и класс для управления процессом |

# Сравнительная характеристика

| **Абстрактная фабрика** | **Фабричный метод** | **Строитель** |
| --- | --- | --- |
| Скрывает реализацию семейства объектов. | Скрывает реализацию объекта. | Скрывает процесс создания объекта, порождает требуемую реализацию |

# характеристика Одиночки

Одиночка может **гарантировать наличие единственного лабиринта** в игре и свободный доступ к нему со стороны всех объектов, не прибегая к глобальным переменным или функциям.

Одиночка также позволяет легко расширить или заменить лабиринт, не трогая существующий код.

# Одиночка (Singleton)

Проблема

Необходимо, чтобы у класса существовал только один экземпляр, к которому нужно создать глобальную точку доступа.

*Результат*

*Сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру.*

# Одиночка (примеры проблем)

В программе может быть довольно много сущностей, которые обязательно должны быть всего в одном экземпляре, например:

- должна быть лишь одна файловая система,
- лишь один файловый менеджер,
- лишь одно соединение к базе данных,
- единственная система ведения системного журнала сообщений
- …

# Одиночка (описание класса)

```
class Singleton {
private:
   int myData;
   static Singleton * instance;
protected:
   Singleton(int data) { myData = data; }
   Singleton() { myData = 0; }
public:
   static Singleton* Singleton::Instance (int data) ;
   static Singleton* Singleton::Instance () ;
   void run(){cout<< "myData="<<myData<<"\n";}
   int getData(){ return myData; }
};
```

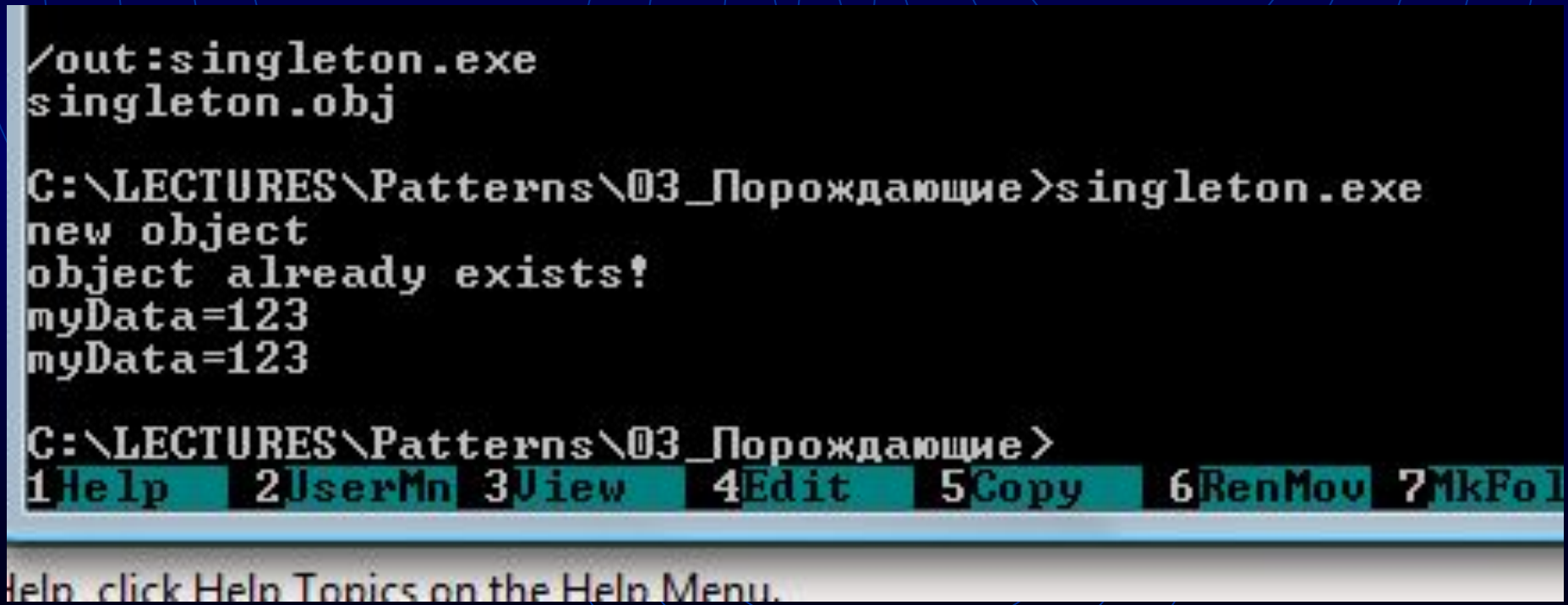
# Одиночка (реализация)

```
Singleton * Singleton::instance = NULL;

Singleton* Singleton::Instance (int data) {
    if (instance == 0) { cout<<"new object \n";
        instance = new Singleton(data);
    }    else cout<<"object already exists!\n";
    return instance;
}
Singleton* Singleton::Instance () {
    if (instance == 0) { cout<<"new object \n";
        instance = new Singleton();
    }    else cout<<"object already exists!\n";
    return instance;
}
```

# Одиночка (работа с объектом)

```
int main(int argc, char *argv[]) {

Singleton * s =  Singleton::Instance (123);
Singleton * d =  Singleton::Instance (456);

s->run();
d ->run();

return 0;
}
```

# Одиночка (результат работы)



# Одиночка (отличия от статического класса)

- у статического класса может быть много точек доступа, в Singleton – одна;
- объект-одиночку можно передать параметром, статический класс – нельзя;
- в случае с Singleton есть возможность контролировать время жизни объекта, в случае со статическим классом – нет;
- объект-одиноку можно сериализировать, статический класс – нельзя.

# Пул объектов (Object Pool)

## Проблема

Создание объекта требует больших затрат или может быть создано только ограниченное количество объектов некоторого класса.

## Решение

Желательно, чтобы все многократно используемые объекты, свободные в некоторый момент времени, хранились в одном и том же пуле объектов. Тогда ими можно управлять на основе единой политики. Для этого класс Object Pool проектируется с помощью паттерна Singleton.

# Пул объектов (Object Pool)

Если у нас много однотипных объектов, создание и уничтожение которых занимает много ресурсов, удобно использовать этот паттерн.

Пулы объектов (известны также как пулы ресурсов) используются для управления кэшированием объектов. Клиент, имеющий доступ к пулу объектов, может избежать создания новых объектов, просто запрашивая в пуле уже созданный экземпляр. Пул объектов может быть растущим, когда при отсутствии свободных создаются новые объекты или с ограничением количества создаваемых объектов. Уничтожение объекта заменяется на его возврат в пул для дальнейшего использования.

# Пул объектов (Object Pool)

*Важный момент:*

Объекты в пуле, которые будут использоваться многократно, должны иметь метод для возврата в своё начальное состояние, чтобы при повторном использовании объекта не было таких данных, которые остались со значениями от прошлого использования.

*Реализация: на основе стека или очереди в зависимости от стратегии выборки объекта из пула и возврата его в пул*

# Задание на лабораторную работу №5
## (порождающие паттерны )

1. Рассмотреть проблему порождения объектов в системе из лабораторной работы 3. Предложить реализацию на основе фабричного метода и/или абстрактной фабрики
2. Внести в систему одиночный объект, который следит за состоянием системы в целом.
3. Ввести для одного из классов возможность клонирования объектов
4. Построить диаграмму классов
5. Представить варианты реализации расширения системы

# Задание на лабораторную работу №6
## (порождающие паттерны )

1. Рассмотреть проблему создания объектов из лабораторной работы 5 на основе паттерна Строитель.
2. Реализовать множественное хранение объектов на основе пула объектов.
3. Выполнить сравнительный анализ реализации порождения объектов из лабораторных работ номер 5 и 6.
4. Построить диаграмму классов
5. Представить варианты реализации расширения системы